Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Носенко Т. І., Машкіна І.В., Яскевич В.О.

Застосування алгоритмів та структур даних у штучному інтелекті

навчальний посібник
для студентів спеціальності «Комп’ютерні науки»

Київ – 2025

УДК 004.85:004.43:

Рекомендовано Вченою радою Факультету інформаційних технологій та математики
(протокол №9 від 15 жовтня 2025 року

Автори: Носенко Т. І., Машкіна І.В., Яскевич В.О.

Рецензенти:
Макаренко Анатолій Олександрович, доктор технічних наук, професор , завідувач кафедри електронних комунікацій та інтернету речей НТУУ КПІ

Ткаченко Ольга Іванівна, к. ф.-м.н., доцент, доцент кафедри інформаційних технологій Навчально-наукового інституту управління, технологій та правових наук, Національний транспортний університет

Носенко Т. І Застосування алгоритмів та структур даних у штучному інтелекті: навчальний посібник / Т.І.Носенко, І.В.Машкіна, В.О.Яскевич Київ : Київський столичний університет імені Бориса Грінченка,
2025. – 201 С

Посібник «*Застосування алгоритмів та структур даних у штучному інтелекті*» є практичним керівництвом для ефективного розв’язання реальних задач програмування. У ньому розглянуто ключові алгоритми (сортування, пошук, графові алгоритми тощо) та структури даних (масиви, списки, дерева, хеш-таблиці), з акцентом на їхню практичну імплементацію, аналіз складності та вибір оптимального рішення для конкретної проблеми зокрема в перспективній галузі штучного інтелекту (ШІ). Видання містить приклади коду, схеми, вправи та типові завдання, що робить його незамінним ресурсом для студентів, розробників та всіх, хто прагне покращити свої навички у конструюванні ефективних і оптимізованих, зокрема для майбутньої роботи в області штучного інтелекту та data science.

УДК 004.85:004.43

.

Зміст

Вступ

Ми живемо в епоху стрімкого розвитку інтелектуальних систем, які трансформують промисловість та повсякденне життя. Ми взаємодіємо з великими мовними моделями, здатними генерувати осмислений текст, користуємося системами персоналізованих рекомендацій та спостерігаємо за прогресом у сфері автономних транспортних засобів. За цими складними технологіями стоїть не магія, а строга наукова дисципліна, в основі якої лежить фундаментальний зв'язок між алгоритмами та структурами даних.

На перший погляд, може здатися, що між текстовим чат-ботом та системою керування автомобілем мало спільного. Проте в їхній основі, на найглибшому рівні, лежить спільний фундамент. Наприклад, ChatGPT є гігантською моделлю, що навчилася передбачати наступне слово в реченні, використовуючи алгоритми на кшталт архітектури "Трансформер" та оптимізаційні методи, як-от градієнтний спуск. Автопілот, у свою чергу, застосовує алгоритми комп'ютерного зору для ідентифікації об'єктів та алгоритми пошуку, подібні до A*, для планування безпечного маршруту. Отже, успіх обох технологій є тріумфом композиції десятків і сотень алгоритмів, кожен з яких ефективно працює зі своїми структурами даних.

Даний навчальний посібник розроблено відповідно до філософії "AI-First", де класичні алгоритми та структури даних розглядаються не як абстрактна теорія, а як основа для створення сучасних інтелектуальних систем. Ви зрозумієте, чому вибір оптимальної структури даних — чи то багатовимірного масиву (тензора) для зберігання параметрів нейронної мережі, чи то графа для моделювання соціальних зв'язків — є критично важливим для продуктивності всієї системи. Ми послідовно розглянемо повний життєвий цикл розробки ШІ-проєкту: від постановки задачі та підготовки даних до навчання й оцінки моделі. На кожному етапі буде продемонстровано, як фундаментальні концепції нашого курсу знаходять своє практичне застосування.

Метою цього посібника є надання ключових навичок для розробки складних систем. Ви навчитеся аналізувати часову та просторову складність, щоб заздалегідь оцінювати життєздатність обраних підходів при роботі з великими обсягами даних. Ви опануєте базові алгоритмічні структури — слідування, розгалуження та повторення — як універсальний інструментарій для побудови будь-якої програмної логіки.

Вивчення цих фундаментальних компонентів є першим і найважливішим кроком для кожного, хто прагне не просто користуватися технологіями майбутнього, а створювати їх. Цей курс закладає теоретичний і практичний фундамент, без якого неможливо побудувати жодну сучасну технологічну систему.

Розділ 1. Базові поняття теорії алгоритмів і структури даних. Фундамент ШІ: від ідеї до коду на Python

1.1. Чому без знання алгоритмів неможливо створити ChatGPT або автопілот?

На перший погляд, між текстовим чат-ботом та системою керування автомобілем мало спільного. Проте в їхній основі, на найглибшому рівні, лежить спільний фундамент — алгоритми та структури даних. Без глибокого розуміння цих концепцій створення таких складних систем було б абсолютно неможливим. Давайте зазирнемо "під капот" кожної з цих технологій.

Випадок 1: ChatGPT (Генеративний Штучний Інтелект)

ChatGPT - це гігантська мовна модель, яка навчилася передбачати наступне найбільш імовірне слово в реченні. Цей, на вигляд простий, процес вимагає колосальної кількості алгоритмічних операцій. В основі лежить алгоритм, відомий як архітектура "Трансформер" - складна система, що використовує механізми "уваги" для зважування важливості різних слів у тексті. Кожен шар цієї архітектури — це набір математичних операцій над величезними матрицями (тензорами).

Щоб "навчити" модель, її "годують" гігантськими обсягами тексту. На цьому етапі працюють алгоритми оптимізації та алгоритми лінійної алгебри. З алгоритмів оптимізації найважливішим є градієнтний спуск та його варіації (Adam). Цей алгоритм ітеративно коригує мільярди параметрів моделі, щоб мінімізувати помилку передбачення. Без нього навчання було б неможливим. Із застосуванням алгоритмів лінійної алгебри все навчання зводиться до надзвичайно швидкого множення матриць, для чого використовуються спеціалізовані алгоритми, оптимізовані для роботи на GPU.

Коли ви ставите питання, модель не просто видає перше-ліпше слово. Вона використовує алгоритми пошуку. Такі алгоритми, як "Променевий пошук" (Beam Search), аналізують не одне, а декілька найбільш імовірних продовжень фрази, щоб вибрати найбільш зв'язний та осмислений варіант.

Усі мільярди параметрів моделі зберігаються у вигляді структур даних - багатовимірних масивів (тензорів). Тексти перетворюються на вектори чисел. Без цих структур даних модель просто не мала б де зберігати свої "знання".

Випадок 2: Автопілот (Комп'ютерний зір та прийняття рішень)

Система автопілота вирішує три основні задачі: "бачити" світ, розуміти його та приймати рішення. Кожна з них — це алгоритми в дії.

1. Зір (сприйняття світу):
    - Алгоритми обробки зображень. Дані з камер обробляються за допомогою згорткових нейронних мереж (CNN). Це специфічний клас алгоритмів, що вміє знаходити патерни в зображеннях (лінії, кути, форми), щоб ідентифікувати автомобілі, людей, дорожні знаки.

- Алгоритми сегментації дозволяють розділити зображення на зони: "дорога", "тротуар", "небо", щоб машина розуміла, де можна їхати.

2. Розуміння (побудова карти світу):
   - Алгоритми об'єднання даних (Sensor Fusion). Автопілот отримує дані з камер, радарів, лідарів. Спеціальні алгоритми (напр., фільтр Калмана) об'єднують ці дані в єдину, більш точну картину світу, відфільтровуючи шуми.
3. Прийняття рішень (дії):
   - Алгоритми планування шляху. Використовуючи побудовану карту оточення, система застосовує алгоритми пошуку, подібні до A* (A-star) або Дейкстри, щоб прокласти безпечний мікро-маршрут в об'їзд перешкод.
   - Алгоритми керування. На основі маршруту та динаміки руху (швидкість, прискорення) алгоритми приймають рішення: повернути кермо на 5 градусів, пригальмувати чи прискоритись.

Карта оточення може бути представлена у вигляді двовимірної сітки (масиву). Об'єкти та їхні властивості зберігаються у вигляді записів або об'єктів. Послідовність дій планується з використанням дерев або графів станів.

Отже, і ChatGPT, і автопілот є тріумфом не однієї технології, а композиції десятків і сотень алгоритмів, кожен з яких ефективно працює зі своїми структурами даних. Саме тому вивчення цих фундаментальних "цеглинок" є першим і найважливішим кроком для кожного, хто хоче не просто користуватися технологіями майбутнього, а створювати їх.

1.2. Етапи розробки ШІ-проектів: від математичної моделі до вибору оптимальної структури даних

Створення системи штучного інтелекту — це не хаотичний творчий акт, а чіткий, послідовний інженерний процес, який має свої етапи. Розуміння цього життєвого циклу дозволяє перетворити бізнес-ідею на працюючий продукт. Кожен із цих етапів нерозривно пов'язаний з фундаментальними поняттями нашого курсу: вибором правильних алгоритмів та оптимальних структур даних.

Рис. 1 . Життєвий цикл ШІ-проекту

Етап 1: Постановка задачі та визначення мети

Це найважливіший етап, на якому закладається фундамент усього проекту. Ми перекладаємо бізнес-проблему на мову машинного навчання. Наприклад:

- Бізнес-проблема - "Клієнти йдуть до конкурентів".
- Задача для ШІ - "Передбачити, який клієнт схильний до відтоку" — це задача класифікації.
- Бізнес-проблема - "Ми хочемо підвищити продажі".
- Задача для ШІ - "Запропонувати користувачу товари, які він, імовірно, купить" — це задача рекомендації.

На цьому етапі ми вперше замислюємося про те, які дані нам потрібні на вході (ознаки клієнта) і що ми хочемо отримати на виході (відповідь "так" чи "ні", список товарів). Це визначає початкові вимоги до наших майбутніх структур даних.

Етап 2: Збір та підготовка даних (Data Collection & Preparation)

Дані — це "паливо" для будь-якого ШІ. Без якісних даних навіть найкращий алгоритм не дасть результату. Збираються дані з різних джерел (бази даних, лог-файли, API). Потім починається найдовший етап — очищення даних (Data Cleaning):

- Заповнення пропущених значень.
- Видалення аномалій та викидів.
- Приведення даних до єдиного формату (нормалізація).

Це є чиста алгоритмізація в дії. Використовуються базові алгоритми (цикли, умовні оператори) та структури даних (списки для зберігання записів, словники для їх представлення), щоб маніпулювати даними.

Етап 3: Вибір та побудова математичної моделі (Model Selection)

На цьому етапі ми обираємо той самий "мозок" нашої системи. На основі поставленої задачі ми обираємо клас алгоритмів, який будемо використовувати. Це і є вибір математичної моделі.

- Для задачі класифікації це може бути Дерево рішень, логістична регресія або нейронна мережа.
- Для передбачення числового значення (напр., ціни квартири) — лінійна регресія.

Вибір моделі безпосередньо диктує, з якими структурами даних та алгоритмами ми будемо працювати.

- Обрали Дерево рішень? Нам потрібно буде реалізувати деревоподібну структуру даних.
- Обрали нейронну мережу? Вся робота буде побудована на операціях з матрицями (тензорами).
- Обрали метод k-найближчих сусідів? Нам знадобиться ефективний алгоритм сортування, щоб швидко знаходити "сусідів".

Етап 4: Навчання моделі (Model Training)

Це процес, під час якого модель "набирається досвіду" на підготовлених даних. Модель пропускає через себе дані та порівнює свої відповіді з правильними. Спеціальний алгоритм оптимізації (найчастіше — градієнтний спуск) крок за кроком коригує внутрішні параметри моделі, щоб зменшити помилку. Це кульмінація застосування складних ітеративних алгоритмів. Ефективність навчання безпосередньо залежить від швидкості виконання цих алгоритмів.

Етап 5: Оцінка та валідація моделі (Model Evaluation)

Після навчання ми повинні переконатися, що наш "мозок" працює коректно, а не просто "зазубрив" відповіді. Модель тестують на даних, яких вона ще не бачила (тестовий набір). Розраховуються метрики якості: точність (accuracy), повнота (recall), точність (precision) тощо. Це є етап чистої алгоритмічної логіки: порівняння результатів, підрахунок правильних та неправильних відповідей, агрегація статистики.

Етап 6: Впровадження та моніторинг (Deployment & Monitoring)

Навіть ідеальна модель марна, якщо нею не можна скористатися. Модель "запаковується" і робиться доступною для кінцевих користувачів, найчастіше у вигляді API. Після цього її роботу постійно відстежують, щоб переконатися, що якість її прогнозів не погіршується з часом. На цьому етапі критично важливою стає часова та просторова складність наших алгоритмів. Якщо модель довго відповідає на запит через неефективний алгоритм або повільну структуру даних, користувачі не будуть нею користуватися.

Отже, як бачимо, розробка ШІ — це інженерний процес, де на кожному кроці ми робимо вибір, що визначає долю всього проекту. І цей вибір завжди зводиться до двох фундаментальних питань: який алгоритм використати для вирішення задачі та в яку структуру даних покласти наші дані для максимальної ефективності.

1.3. Поняття алгоритму: Класичний фундамент у світі штучного інтелекту

Ми часто говоримо про "алгоритми YouTube" чи "алгоритми Instagram", маючи на увазі складні, майже магічні системи, що керують нашим цифровим життям. Проте, щоб зрозуміти, як вони працюють, нам потрібно повернутися до основ — до класичного визначення алгоритму. Виявляється, що ці фундаментальні принципи, сформульовані десятиліття тому, є як ніколи актуальними в епоху ШІ, виступаючи надійним каркасом для найсміливіших інновацій.

Частина 1: Класичні основи

1. Що таке алгоритм?

В найпростішому розумінні, **алгоритм** — це скінченна, чітко визначена послідовність кроків або правил, призначена для вирішення конкретної задачі. Це, по суті, детальний рецепт або інструкція зі збирання: візьміть ці інгредієнти (вхідні дані), виконайте ці дії в такій послідовності, і ви гарантовано отримаєте очікуваний результат (вихідні дані). Подібно до того, як нотний запис дозволяє відтворити музичний твір, алгоритм є універсальною мовою, що дозволяє виконавцю відтворити процес вирішення задачі.

2. Хто є виконавцем?

Кожен алгоритм призначений для певного виконавця — системи або особи, що здатна розуміти та беззаперечно виконувати його кроки.

- Класичний виконавець - це центральний процесор (CPU) комп'ютера, який послідовно, крок за кроком, виконує інструкції машинного коду. Це надійний, але часто повільний для певних задач працівник.
- Виконавець у світі ШІ - це значно складніша система, оптимізована для паралелізму. Виконавцем може бути графічний процесор (GPU), що, наче тисячі одночасних працівників, паралельно обробляє тисячі простих операцій над даними. Або ж це може бути ціла програмна екосистема (як TensorFlow чи PyTorch), яка керує розподіленими обчисленнями на кластері серверів. Але суть залишається незмінною: виконавець чітко слідує закладеним у модель правилам, незалежно від його складності.

3. Ключові властивості алгоритму

Щоб послідовність інструкцій вважалася алгоритмом, вона повинна відповідати кільком ключовим властивостям. Порушення хоча б однієї з них перетворює чіткий план на беззмістовний набір дій.

- Дискретність (Крововість): Алгоритм складається з окремих, завершених кроків. Неможливо виконати "півтора кроку". Кожна дія чітко відокремлена від наступної. *Якщо ця властивість порушена, система опиняється в невизначеному стані, що призводить до хаосу та непередбачуваних помилок.*
- Визначеність (Детермінованість): Кожен крок алгоритму має бути чітким і однозначним. При однакових вхідних даних алгоритм завжди повинен давати однаковий результат. *(Примітка: у ШІ існують стохастичні алгоритми, але навіть у них випадковість є контрольованою і відтворюваною). Якщо ця властивість порушена, алгоритм стає ненадійним, а його результатам не можна довіряти.*
- Скінченність: Алгоритм має гарантовано завершити свою роботу за скінченну кількість кроків. Нескінченний цикл — це помилка в алгоритмі. *Якщо ця властивість порушена, програма "зависає", безкінечно споживаючи ресурси і так ніколи й не даючи результату.*
- Масовість: Алгоритм має бути застосовним не для однієї унікальної задачі, а для цілого класу однотипних задач (наприклад, алгоритм сортування повинен працювати для будь-якого масиву чисел, а не тільки для [3, 1, 2]). *Якщо ця властивість порушена, ми отримуємо не універсальний інструмент, а одноразове рішення, яке неможливо масштабувати.*
- Результативність (Коректність): Після завершення роботи алгоритм повинен давати правильний результат, що відповідає поставленій задачі. Якщо ця властивість порушена, алгоритм, навіть будучи швидким та ефективним, є абсолютно марним, бо виробляє "сміття".

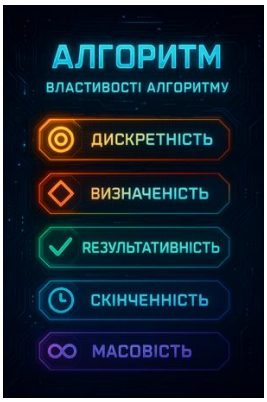


Рис.2. Властивості алгоритмів

Частина 2: Як ці поняття пов'язані з етапами розробки ШІ-проектів?

А тепер давайте подивимось, як ці "сухі" академічні властивості проявляються на кожному етапі створення реального ШІ-продукту.

***Таблиця** 1. Етапи розробки ШІ-проекту*

Етап розробки ШІ	**Практичне значення властивостей в ШІ**
1. Постановка задачі	На цьому етапі ключовою є Визначеність. Ми повинні чітко визначити, що саме має робити наш алгоритм. "Покращити бізнес" — це не визначена мета. "Передбачити з імовірністю >80%, чи піде клієнт у відтік протягом наступних 3 місяців на основі історії його покупок" — це чітка, детермінована задача для алгоритму.
2. Підготовка даних	Тут панує Дискретність. Процес очищення даних — це чітка послідовність кроків: 1) прочитати файл, 2) видалити пусті рядки, 3) заповнити пропущені значення середнім, 4) нормалізувати дані. Помилка на одному кроці каскадом поширюється на всі наступні. Масовість також важлива: наш скрипт для очищення має працювати для будь-якого набору даних схожого формату, а не лише для тестового файлу.
3. Навчання моделі	Навчання — це, по суті, один великий ітеративний алгоритм (напр., градієнтний спуск). Тут критичною є Скінченність. Процес навчання має колись завершитись (або після досягнення потрібної точності, або після заданої кількості епох). Якби він був нескінченним, ми б ніколи не отримали готову модель, а лише витратили б тисячі доларів на хмарні обчислення.
4. Оцінка моделі	На цьому етапі ми перевіряємо Результативність. Чи дає наш алгоритм правильні відповіді на даних, яких він не бачив? Алгоритм може бути швидким і завершуватися, але якщо він некоректний (напр., завжди передбачає один і той самий клас), він не має цінності. Ми використовуємо формальні метрики (точність, повнота), щоб об'єктивно виміряти його коректність.
5. Впровадження	У реальному продукті, як-от автопілот, найважливішими стають Скінченність та Результативність у жорстких часових рамках. Алгоритм розпізнавання пішохода має не просто спрацювати правильно, а зробити це за мілісекунди. Порушення скінченності тут є катастрофічним. Визначеність також критична: реакція на той самий дорожній знак має бути завжди однаковою.

Отже, хоча сучасні моделі ШІ є надзвичайно складними і можуть здаватися "чорними скриньками", вони, як і найпростіші програми, підкоряються фундаментальним законам алгоритмізації. Розуміння цих класичних властивостей — це те, що відрізняє інженера

від простого користувача. Воно дозволяє будувати надійні, передбачувані та ефективні системи штучного інтелекту, а не просто сподіватися на "магію". Це фундамент, без якого неможливо звести жоден технологічний хмарочос.

1.4. Мова ідей та інструкцій: Способи опису алгоритмів

Алгоритм — це, по суті, ідея, чітко сформульований план дій. Але як передати цю ідею іншим або, що найважливіше, комп'ютеру? Для цього існують різні способи опису, кожен з яких має свої переваги та недоліки. Вони утворюють своєрідний спектр: від максимально абстрактного та зрозумілого для людини до абсолютно точного та конкретного для машини. Вибір способу залежить від мети: пояснити ідею колезі за кавою, спланувати архітектуру складного модуля чи дати безпосередню команду для виконання.

1. Словесний (природномовний) опис

Це найпростіший та найбільш інтуїтивний спосіб. Алгоритм описується звичайною людською мовою у вигляді послідовності речень, наче ви пояснюєте рецепт або дорогу другові. Це - словесна інструкція, кулінарний рецепт, покрокове керівництво. Наприклад, *алгоритм знаходження найбільшого числа в списку.*" Спочатку припустимо, що перше число у списку є найбільшим, і запам'ятаємо його в окремій змінній. Потім послідовно переглянемо кожне наступне число в списку, від другого до останнього. Якщо чергове число більше за те, яке ми запам'ятали, то забудемо старе і запам'ятаємо нове як найбільше. Коли список закінчиться, те число, яке ми запам'ятали останнім, і буде результатом."

- Переваги:
  - Легко зрозуміти будь-якій людині, навіть без технічних знань.
  - Ідеально підходить для початкового обговорення ідеї та мозкового штурму.
- Недоліки:
  - Неоднозначність: Природна мова сповнена синонімів, метафор та неточностей. Фраза "взяти наступний елемент" може тлумачитися по-різному. Це може призвести до неправильної реалізації та помилок, які важко відстежити.
  - Багатослівність: Опис навіть простих дій може бути громіздким та займати багато місця, що ускладнює сприйняття складної логіки.
  - Непридатність для прямої реалізації.

2. Псевдокод

Це напівформальний спосіб опису, який є "мостом" між людською мовою та мовою програмування. Псевдокод використовує структуру та ключові слова мов програмування (if-then-else, for, while, return), але не має строгого синтаксису і може містити фрази звичайною мовою. Він дозволяє сфокусуватися на логіці, ігноруючи синтаксичні деталі, як-от крапки з комою чи типи даних. Це - структурований план, "чернетка" програми, скелет майбутнього коду. Наприклад, *алгоритм знаходження найбільшого числа в списку.*

```
FUNCTION знайти_максимум(список):
    ЯКЩО список порожній:
        ПОВЕРНУТИ помилку "Список порожній"
    максимум = список[0]
    ДЛЯ кожного елемента В списку, починаючи з другого:
        ЯКЩО елемент > максимум:
            максимум = елемент
    ПОВЕРНУТИ максимум
```

Переваги:

- Структурованість: Чітко показує логіку, цикли та розгалуження, усуваючи неоднозначність словесного опису.
- Компактність: Значно коротший за словесний опис.
- Незалежність від мови: Легко перекладається на будь-яку мову програмування (Python, C++, Java), слугуючи універсальним планом для розробників.

- Недоліки:
  - Вимагає базового розуміння принципів програмування.
  - Не є стандартизованим (різні люди можуть писати псевдокод трохи по-різному).

3. Графічний опис (блок-схеми)

Це візуальний спосіб представлення алгоритму за допомогою стандартного набору геометричних фігур, з'єднаних стрілками, що показують потік виконання. Цей метод був дуже популярним на зорі програмування. **Це -в**ізуальна карта логіки алгоритму, діаграма процесу. Наприклад, алгоритм знаходження найбільшого числа в списку.

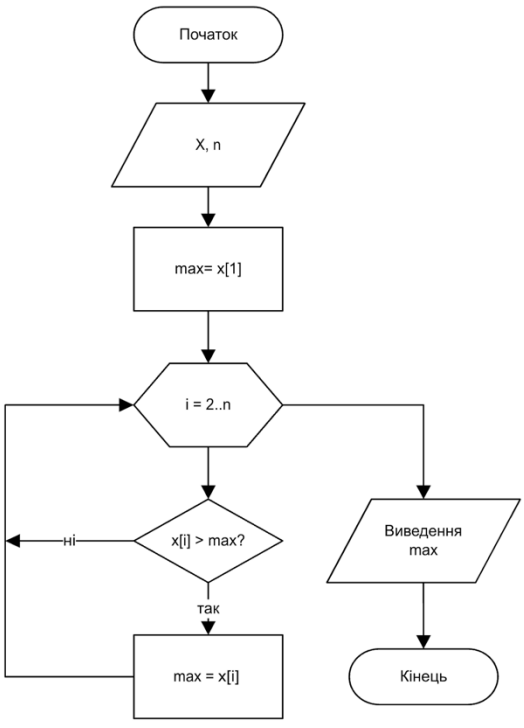


Рис.3. Блок-схема для знаходження максимального елемента в масиві

- Переваги:

- Наочність: Дуже легко простежити логічний потік, особливо для людей з візуальним мисленням. Чудово підходить для презентацій та пояснення простих алгоритмів.
- Стандартизація: Форми блоків є загальноприйнятими (овал — початок/кінець, прямокутник — дія, ромб — умова/рішення), що робить схеми зрозумілими на міжнародному рівні.

- Недоліки:
  - Громіздкість: Для складних алгоритмів блок-схеми стають величезними, заплутаними і їх важко малювати та змінювати. Навіть незначна зміна в логіці може вимагати повного перемальовування схеми. Це робить їх непрактичними для сучасної гнучкої (Agile) розробки, де вимоги та логіка можуть часто змінюватися.
  - У сучасній гнучкій розробці використовуються рідко, переважно в навчальних цілях або для документації дуже високого рівня.

4. Мова програмування

Це найбільш формальний, точний і детальний спосіб опису алгоритму, призначений безпосередньо для виконавця-комп'ютера. Це фінальна стадія перетворення ідеї на працюючий продукт.

Що це? Фінальна, готова до виконання інструкція.

Приклад: *Алгоритм знаходження найбільшого числа в списку на Python.*

```
# Створюємо список чисел
my_numbers = [15, 27, 8, 42, 11, 98, 56]
print(f"Початковий список: {my_numbers}")


# Перевіряємо, чи список не порожній
if not my_numbers:
    print("Список порожній.")
else:
    # Припускаємо, що перший елемент є найбільшим
    largest_so_far = my_numbers[0]

    # Проходимо по кожному числу в списку, починаючи з другого
    for number in my_numbers[1:]:
        # Якщо поточне число більше, ніж те, яке ми вважали найбільшим досі...
        if number > largest_so_far:
            # ...оновлюємо наше "найбільше" число
            largest_so_far = number

    print(f"Найбільше число, знайдене вручну: {largest_so_far}")
```

```
Початковий список: [15, 27, 8, 42, 11, 98, 56]
Найбільше число, знайдене вручну: 98
```

Рис. 4. Приклад реалізації алгоритму мовою програмування

Переваги:

- Абсолютна точність: Не залишає місця для неоднозначності. Кожна команда має лише одне тлумачення.
- Виконуваність: Може бути прямо запущений на комп'ютері для перевірки та отримання результату.

Недоліки:

- Прив'язка до синтаксису: Зрозумілий лише тим, хто знає конкретну мову програмування.
- Може приховувати загальну логіку за деталями реалізації. Наприклад, один рядок коду з викликом бібліотечної функції може приховувати під собою дуже складний алгоритм, що ускладнює розуміння саме *методу* вирішення задачі.

***Таблиця** 2. Порівняння способів опису алгоритмів*

Спосіб опису	**Основне призначення**	**Переваги**	**Недоліки**
Словесний	Початкове обговорення ідеї	Легкість для неспеціалістів	Неточність, багатослівність
Псевдокод	Планування та проектування	Структура, компактність, незалежність	Не є стандартом, вимагає знань
Блок-схема	Візуалізація логіки	Наочність, стандартні символи	Громіздкість для складних задач
Мова програмування	Реалізація та виконання	Точність, виконуваність	Прив'язка до синтаксису, складність

У нашому курсі ми будемо найчастіше використовувати зв'язку Блок-схема -> Мова програмування (Python). Блок-схема допоможе нам спланувати логіку наших ШІ-моделей, не заглиблюючись у синтаксис, а Python — швидко та ефективно втілити її в життя. Опанування цього переходу від абстрактної логіки до конкретної реалізації є однією з ключових навичок успішного розробника.

Уніфікація способів опису алгоритмів, хоча б у вигляді загальноприйнятих конвенцій, є ключовою з кількох причин:

1. Однозначність та ясність: Стандартизований опис (особливо псевдокод та блок-схеми) усуває двозначність, яка притаманна природній мові. Це гарантує, що різні розробники та дослідники зрозуміють логіку алгоритму однаково.
2. Спрощення комунікації та співпраці: Коли команда розробників використовує єдиний стиль опису, це значно полегшує обговорення, перевірку та спільну роботу над алгоритмами. Не потрібно витрачати час на "переклад" з одного стилю на інший.
3. Порівняння та аналіз: Уніфікований опис дозволяє легше порівнювати різні алгоритми між собою, аналізувати їхню ефективність (наприклад, за допомогою О-нотації) та знаходити переваги чи недоліки кожного з них.

4. Навчання та документація: В освітньому процесі та в технічній документації стандартизований підхід (найчастіше — псевдокод) є найкращим способом передати знання про те, як працює алгоритм, не прив'язуючись до конкретної технології.

На відміну від багатьох інженерних галузей, у програмуванні немає єдиного, глобального та суворого міжнародного стандарту (ISO), який би наказував, як саме описувати всі алгоритми. Однак існують стандарти та загальноприйняті практики для окремих способів опису.

Для блок-схем існує міжнародний стандарт ISO 5807:1985 ("Information processing — Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts"). В Україні його аналогом є ДСТУ 19.701-90 «ЄСПД. Схеми алгоритмів, програм, даних і систем. Позначення умовні і правила виконання», який встановлює єдині графічні позначення для елементів блок-схем. Саме тому ромби для умов та прямокутники для дій виглядають однаково у більшості підручників у всьому світі.

Для псевдокоду не існує єдиного суворого стандарту. Однак склалися загальноприйняті конвенції, що базуються на синтаксисі Паскале-подібних мов (з використанням ключових слів якщо, то, інакше, цикл, поки тощо). Головне правило — псевдокод має бути інтуїтивно зрозумілим.

Для аналізу ефективності стандартом де-факто є асимптотична нотація (О-нотація, Big O notation), яка уніфікує спосіб оцінки часової та просторової складності алгоритмів.

Отже,хоча не існує єдиного документа, який би регулював усі способи опису алгоритмів, індустрія та академічна спільнота виробили потужні конвенції та стандарти для окремих аспектів (графічне представлення, аналіз складності). Уніфікація, навіть на рівні таких конвенцій, є критично важливою, оскільки вона забезпечує ясність, точність та ефективну комунікацію — фундамент для будь-якої інженерної дисципліни.

1.5. Оцінка часової і просторової складності: як зрозуміти, чи "потягне" ваш алгоритм гігабайти даних

Уявіть, що у вас є два рецепти для приготування вечері. Один — простий і зрозумілий, але вимагає постійно бігати до холодильника за кожним інгредієнтом окремо. Інший — трохи складніший, але радить одразу викласти всі інгредієнти на стіл. Для вечері на двох людей різниця буде непомітною. А тепер уявіть, що ви готуєте бенкет на тисячу гостей. Перший "алгоритм" перетвориться на катастрофу, тоді як другий дозволить впоратись із завданням.

Оцінка складності алгоритму — це спосіб заздалегідь зрозуміти, як поведе себе наш "рецепт" (алгоритм) при збільшенні "кількості гостей" (обсягу даних). У світі ШІ, де ми маємо справу з гігабайтами і навіть терабайтами даних, цей аналіз є критично важливим, бо він відокремлює теоретично можливі проекти від практично реалізованих.

Часова складність - "скільки часу це займе?"

Часова складність — це не точний час у секундах (бо він залежить від потужності комп'ютера), а оцінка того, як зростає кількість операцій, яку виконує алгоритм, при збільшенні розміру вхідних даних (n). Для вимірювання використовується нотація "Великого О" (Big O Notation), яка описує найгірший сценарій. Чому саме найгірший? Тому що нам потрібна гарантія. Ми повинні знати максимальну межу, вище якої час роботи точно не підніметься, щоб система залишалася стабільною та передбачуваною навіть за найнесприятливіших умов.

Розглянемо найпоширеніші класи складності.

- O(1) — Константна складність: Час виконання не залежить від розміру даних. Це ідеал, до якого прагнуть.
  - Приклад: Взяти елемент масиву за індексом (my_list[5]) або отримати значення зі словника (хеш-таблиці) за ключем. Неважливо, скільки в масиві елементів, 10 чи 10 мільйонів, ця операція завжди займає однаковий час.
  - Аналогія: Взяти першу книгу з полиці або книгу, номер якої вам відомий.
- O(log n) — Логарифмічна складність: Час виконання зростає дуже повільно. З кожним кроком алгоритм відкидає половину даних, що робить його надзвичайно ефективним для великих обсягів.
  - Приклад: Бінарний пошук у відсортованому масиві. Якщо дані не відсортовані, цей метод не працює.
  - Аналогія: Знайти слово у паперовому словнику. Ви відкриваєте його посередині і одразу відкидаєте половину сторінок, значно звужуючи область пошуку.
- O(n) — Лінійна складність: Час виконання зростає прямо пропорційно кількості даних. Це дуже поширений і загалом прийнятний рівень складності.
  - Приклад: Пошук елемента у невідсортованому списку (потрібно переглянути кожен елемент) або обчислення середнього значення всіх елементів масиву.
  - Аналогія: Прочитати назву кожної книги на полиці, щоб знайти потрібну.
- O(n log n) — Лінійно-логарифмічна складність: Трохи гірше за лінійну, але все ще вважається дуже ефективною. Це "золотий стандарт" для алгоритмів сортування.
  - Приклад: Ефективні алгоритми сортування (сортування злиттям, швидке сортування).
  - Аналогія: Дуже ефективний спосіб розкласти книги на полиці за абеткою, розділяючи їх на стоси, сортуючи кожен стос, а потім об'єднуючи.
- O(n^2) — Квадратична складність: Час виконання зростає пропорційно квадрату від кількості даних. Такі алгоритми стають дуже повільними вже на відносно невеликих наборах даних. Це "червона зона" для роботи з великими даними.
  - Приклад: Прості алгоритми сортування (бульбашкою, вибором), де кожен елемент порівнюється з кожним. Наслідки цього катастрофічні: для сортування 100 елементів потрібно ~10 000 операцій, а для 10 000 елементів — вже ~100 000 000 операцій!
  - Аналогія: Взяти кожну книгу і порівняти її з кожною іншою книгою на полиці, щоб знайти пари однакових.
- O(2^n) — Експоненційна складність: "Вибухова" складність. Такі алгоритми стають непридатними навіть для невеликої кількості даних (напр., n > 30).
  - Приклад: Розв'язання задачі комівояжера повним перебором. Навіть для 20 міст кількість можливих маршрутів перевищує кількість піщинок на Землі, що робить такий підхід абсолютно нереалістичним.
  - Аналогія: Спробувати всі можливі комбінації розташування книг на полиці, щоб знайти "ідеальну".

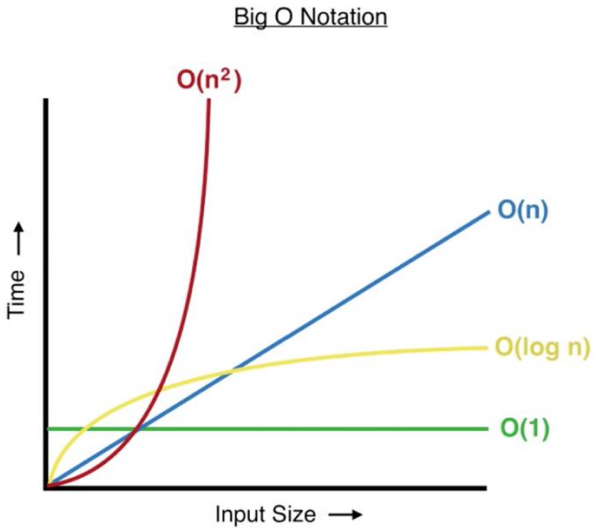


Рис. 5. Зростання функцій O(1), O(log n), O(n), O(n log n), O(n²)

Просторова складність -"Скільки пам'яті це займе?"

Просторова складність — це оцінка того, як зростає обсяг додаткової пам'яті, який потрібен алгоритму, при збільшенні розміру вхідних даних (n). Часто існує компроміс: можна прискорити алгоритм, але за рахунок пам'яті. Наприклад, можна заздалегідь обчислити та зберегти в пам'яті всі можливі результати (створити 'кеш' або таблицю пошуку). Тоді отримання відповіді буде миттєвим (O(1)), але це може вимагати гігабайтів пам'яті. І навпаки, можна економити пам'ять, обчислюючи результат щоразу заново, але це буде повільніше.

- O(1) — Константна складність: Алгоритм використовує фіксовану кількість пам'яті, незалежно від розміру вхідних даних (напр., кілька змінних для лічильників чи індексів).
- O(n) — Лінійна складність: Алгоритму потрібно додатково стільки пам'яті, скільки елементів у вхідних даних (наприклад, для створення копії масиву). Сортування злиттям часто потребує O(n) додаткової пам'яті для тимчасових масивів.

Як це пов'язано з навчанням ШІ-моделей?

Уявімо, що нам потрібно навчити модель на датасеті з 10 мільйонів (10^7) зображень.

1. Часова складність:
   - Якщо ми оберемо алгоритм попередньої обробки зі складністю $O(n^2)$, кількість операцій буде приблизно $(10^7)^2 = 10^{14}$. Сучасний комп'ютер виконує близько 10^9 операцій на секунду. Отже, цей процес займе $10^{14} / 10^9 = 100\ 000$ секунд, або більше доби! В умовах хмарних обчислень це означає не тільки втрачений час, але й значні фінансові витрати, що робить проект економічно невигідним.
   - Якщо ж ми оберемо алгоритм зі складністю O(n log n), кількість операцій буде приблизно $10^7 * \log(10^7) \approx 10^7 * 23 \approx 2.3 * 10^8$. Це займе менше секунди.
   - Висновок: Для навчання моделей на гігабайтах даних підходять лише алгоритми з лінійною або лінійно-логарифмічною складністю. Саме тому в основі бібліотек для ШІ лежать високооптимізовані алгоритми.
2. Просторова складність:

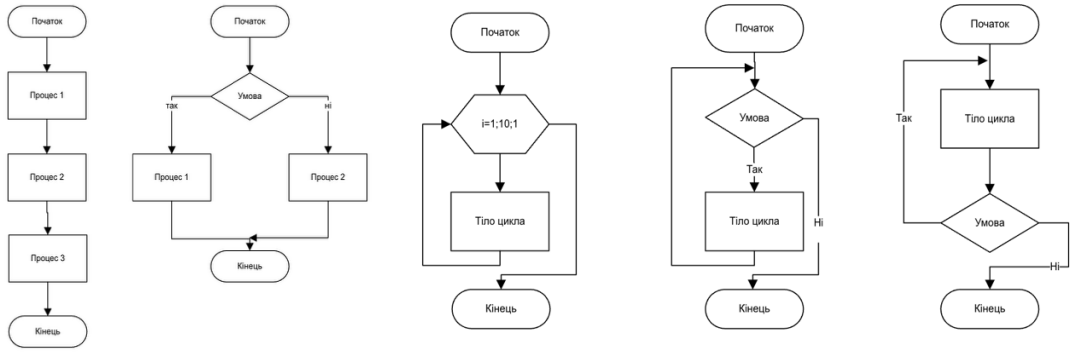
- Сучасні великі мовні моделі (LLM), як-от GPT, мають мільярди параметрів. Кожен параметр — це число, що займає місце в пам'яті.
- Якщо модель має 175 мільярдів параметрів, і кожен зберігається як 16-бітне число (2 байти), то лише для зберігання самої моделі потрібно $175 * 10^9 * 2 = 350 * 10^9$ байт = 350 ГБ пам'яті! Це перевищує обсяг оперативної пам'яті більшості серверів.
- Висновок: Це пояснює, чому для навчання та роботи великих моделей потрібні потужні GPU з величезним обсягом відеопам'яті (VRAM), яка працює значно швидше за звичайну RAM для паралельних обчислень. Саме просторова складність моделі диктує екстремальні вимоги до "заліза" і стимулює розробку технік оптимізації, як-от квантизація моделі (зменшення точності представлення чисел для економії пам'яті).

Розуміння часової та просторової складності — це не академічна вправа. Це ключова інженерна навичка, яка дозволяє заздалегідь оцінити, чи є обраний вами підхід життєздатним. Вона допомагає зрозуміти, чому одні технології "злітають", а інші залишаються нереалізованими, і як обрати правильні алгоритми та структури даних, щоб ваш проект не "завис", зіткнувшись із реальним обсягом даних. Це не просто теорія, а основа інженерної інтуїції, яка дозволяє приймати правильні архітектурні рішення ще до написання першого рядка коду, відокремлюючи життєздатні ідеї від тих, що приречені на провал під вагою реальних даних.

1.6. Типові (базові) алгоритмічні структури

Уявіть, що ви хочете побудувати будь-яку можливу будівлю — від простого будиночка до грандіозного хмарочоса. Незважаючи на всю різноманітність архітектури, в основі всього лежать одні й ті самі базові елементи: цеглини, балки, перекриття. Розуміючи, як їх поєднувати, ви можете створити будь-що.

В програмуванні ситуація дуже схожа. Згідно з теоремою про структурне програмування, доведеною Коррадо Бьомом та Джузеппе Якопіні, будь-який, навіть найскладніший, алгоритм можна побудувати, використовуючи лише три базові керуючі структури. Опанувавши їх, ми отримуємо універсальний конструктор для створення будь-якої програмної логіки, чи то простий калькулятор, чи то складна нейронна мережа.



а *б* *в* *г* *д*

Рис.6. Базові алгоритмічні структури: а - слідування, б - розгалуження, в – цикл з лічильником, г – цикл з переддумовою, д - цикл з післяумовою

1. Слідування (Sequence)

Це найпростіша та найбільш фундаментальна структура, яка є невидимим каркасом для всіх інших. Це лінійна, послідовна дія, інструкції якої виконуються одна за одною, суворо в тому порядку, в якому вони записані, без пропусків чи повернень назад. Це потік виконання за замовчуванням. Наприклад, ранковий ритуал. Ви прокидаєтесь (дія 1), потім вмиваєтесь (дія 2), потім снідаєте (дія 3). Порядок дій є критично важливим. Якщо ви спробуєте спочатку поснідати, а потім прокинутись, система зазнає "логічного збою". Хоча ця структура здається очевидною, саме вона забезпечує передбачуваність програми. Порушення послідовності — одна з найчастіших причин помилок. Наприклад, спроба використати змінну до того, як їй було присвоєно значення, є класичним порушенням принципу слідування.

Python

```
# Дія 1: Отримати вхідні дані для обробки
print("--- Початок обробки даних ---")
source_file = "raw_data.txt"
data = read_data_from_file(source_file) # Припустимо, ця функція існує

# Дія 2: Очистити дані
cleaned_data = clean_data(data) # Видалити помилки, заповнити пропуски

# Дія 3: Проаналізувати дані
analysis_result = analyze_data(cleaned_data) # Обчислити статистику

# Дія 4: Вивести результат
print(f"Результат аналізу: {analysis_result}")
print("--- Обробку завершено ---")
```

2. Розгалуження (Selection / Branching)

Ця структура додає в алгоритм здатність "приймати рішення" та обирати різні шляхи виконання залежно від певних умов. Вона перетворює лінійний шлях на перехрестя. Це є вибір одного з двох або більше варіантів дій на основі перевірки логічної умови (яка може бути істинною True або хибною False). Ця умова є "дорожнім знаком", що вказує,

куди рухатись далі. наприклад, вибір одягу перед виходом на вулицю. *Якщо* на вулиці йде дощ, *то* взяти парасольку. *Інакше, якщо* на вулиці сонячно, *то* взяти сонячні окуляри. *В іншому випадку* (якщо хмарно) не брати нічого.

- Основні форми:
  - if (неповне розгалуження): "Дія за особливих обставин". Дія виконується, тільки якщо умова істинна. Якщо ні — цей блок коду просто ігнорується.
  - if-else (повне розгалуження): "Вибір на роздоріжжі". Обирається один з двох шляхів, і програма гарантовано виконає або блок if, або блок else.
  - if-elif-else (множинне розгалуження): "Вибір з кількох варіантів". Дозволяє перевірити кілька умов послідовно, доки не буде знайдено першу істинну.
- Приклад в контексті AI. Уявіть простий фільтр для електронної пошти.

```
# Перевіряємо листа на ознаки спаму
email_subject = "Безкоштовний приз! Ви виграли!"
sender_is_known = False

if "безкоштовний приз" in email_subject or "виграли" in email_subject:
  print("Статус: СПАМ (підозрілі слова в темі)")
elif not sender_is_known:
  print("Статус: ПІДОЗРІЛИЙ (невідомий відправник)")
else:
  print("Статус: Звичайний лист")
```

Ця структура є робочою конячкою програмування. Вона дозволяє виконувати один і той самий блок коду багаторазово, що є основою для обробки великих обсягів даних, навчання моделей та будь-яких ітеративних процесів.

3. Цикл - це багаторазове виконання певної послідовності дій, доки виконується певна умова або поки не буде пройдено всі елементи набору. Наприклад, миття посуду. Ви повторюєте дію "мити тарілку" *доти*, *доки* в раковині не залишиться брудного посуду. Кількість тарілок може бути різною, але сам процес повторюється.

- Основні форми та коли їх використовувати:
  - Цикл з лічильником (for): Використовується, коли кількість повторень відома заздалегідь або коли потрібно перебрати всі елементи колекції (списку, рядка). Це найбезпечніший тип циклу, оскільки він гарантовано завершиться.
  - Цикл з умовою (while): Використовується, коли кількість повторень невідома, і цикл має тривати, доки умова залишається істинною. Тут криється небезпека нескінченного циклу, якщо забути змінити змінну, від якої залежить умова.

Python

```
# Приклад циклу for: обробити список зображень
image_files = ["img1.jpg", "img2.png", "img3.jpg"]
processed_images = 0
for filename in image_files:
  print(f"Обробка файлу {filename}...")
  # Тут міг би бути складний код обробки
  processed_images += 1
print(f"Всього оброблено {processed_images} зображень.")
```

```
# Приклад циклу while: очікування підтвердження від користувача
user_input = ""
while user_input.lower() != "так":
    user_input = input("Ви підтверджуєте дію? (введіть 'так' для продовження):
")
print("Дію підтверджено. Продовжуємо...")
```

Справжня потужність програмування розкривається, коли ми починаємо комбінувати та вкладати ці три структури одна в одну. Наприклад, ми можемо всередині циклу for (повторення) поставити умову if-else (розгалуження), яка для кожного елемента буде виконувати послідовність дій (слідування). Саме так і будуються складні алгоритми — від простих скриптів до нейронних мереж.

Розглянемо комплексний приклад:
Завдання: Проаналізувати список студентів. Потрібно відібрати тих, хто має високий бал (>=90) для участі в олімпіаді, та відрахувати тих, хто має незадовільний (< 50).

Python

```
students = [
    {"name": "Анна", "grade": 95},
    {"name": "Богдан", "grade": 48},
    {"name": "Віктор", "grade": 75},
    {"name": "Галина", "grade": 91}
]

olympiad_candidates = []
to_expel = []

# 1. Повторення: перебираємо кожного студента у списку
for student in students:
    # 2. Слідування: отримуємо ім'я та оцінку для поточного студента
    name = student["name"]
    grade = student["grade"]
    print(f"Аналіз студента: {name}...")

    # 3. Розгалуження: приймаємо рішення на основі оцінки
    if grade >= 90:
        print(f"  -> Кандидат на олімпіаду.")
        olympiad_candidates.append(name)
    elif grade < 50:
        print(f"  -> Кандидат на відрахування.")
        to_expel.append(name)
    else:
        print(f"  -> Звичайна успішність.")

# 4. Слідування: виводимо підсумкові списки
print("\n--- Результати аналізу ---")
print("Кандидати на олімпіаду:", olympiad_candidates)
print("Кандидати на відрахування:", to_expel)
```

Цей простий приклад демонструє, як комбінація трьох базових структур дозволяє реалізувати досить складну логіку обробки даних.

Отже, слідування, розгалуження та повторення — це три кити, на яких стоїть уся алгоритмізація. Розуміння того, як і коли застосовувати кожну з цих структур, є фундаментальною навичкою, що дозволяє розбивати будь-яку складну задачу на прості, зрозумілі та реалізовані кроки. Це не просто ваш алфавіт, а фундаментальна граматика створення. Опанувавши її, зможемо писати не просто окремі 'речення', а цілі 'твори' — складні та ефективні програми, що вирішують реальні задачі.

1.7. Архітектура інформації: Поняття структури даних та рівні її опису

Якщо алгоритми — це "мозок", що приймає рішення, то структури даних — це його пам'ять. Але це не просто пасивне, хаотичне сховище. Це добре організована система, архітектура якої безпосередньо впливає на те, наскільки швидко мозок може знаходити, обробляти та зберігати потрібну інформацію. Неправильно організована пам'ять може зробити навіть найгеніальніший мозок повільним та неефективним, змушуючи його витрачати дорогоцінні ресурси на пошук замість мислення.

Поняття структури даних

Структура даних — це спосіб організації, зберігання та керування даними, що не тільки визначає логічні зв'язки між ними, але й забезпечує ефективний доступ та модифікацію. Простіше кажучи, це контейнер для даних, який надає "інтерфейс" для роботи з ними.

Уявіть собі бібліотеку. Усі книги звалені в одну величезну купу в центрі зали. Щоб знайти потрібну, вам доведеться перебрати кожну книгу. Це є приклад поганої організації структур даних. Хороший приклад структури даних, якщо книги розставлені на полицях (секціях), відсортовані за жанром, а в межах жанру — за прізвищем автора в алфавітному порядку. Додатково існує електронний каталог (індекс), де ви можете миттєво знайти номер полиці за назвою книги.

В обох випадках набір даних (книги) однаковий. Але в другому випадку, завдяки структурі, будь-який алгоритм пошуку книги (навіть найпростіший) буде в тисячі разів ефективнішим.

Алгоритми та структури даних нерозривні. Це дві сторони однієї медалі. Алгоритми оперують даними, а структури даних надають ці дані у зручному для оперування вигляді. Навіть найшвидший алгоритм сортування (O(nlogn)) буде працювати жахливо повільно, якщо застосувати його до структури даних, де доступ до кожного елемента є повільним (наприклад, зв'язний список). Вибір неправильної структури може звести нанівець переваги найшвидшого алгоритму.

Як це пов’язано зі світом ШІ? Параметри нейронної мережі — це мільярди чисел. Якби вони зберігалися хаотично, жоден алгоритм навчання не зміг би з ними працювати. Але вони організовані у чітку структуру — тензор (багатовимірний масив), який представляє

собою шари, нейрони та зв'язки. Саме ця структура дозволяє не тільки логічно їх організувати, але й ефективно виконувати паралельні обчислення на GPU. Алгоритм зворотного поширення помилки (backpropagation) покладається на цю матричну структуру для швидкого та ефективного обчислення градієнтів та оновлення ваг.

Рівні опису даних (Абстракція)

Щоб керувати складністю, програмісти та інженери думають про структури даних на різних рівнях абстракції. Це дозволяє сфокусуватися на важливих аспектах, не заглиблюючись у деталі, які на даному етапі не є суттєвими.

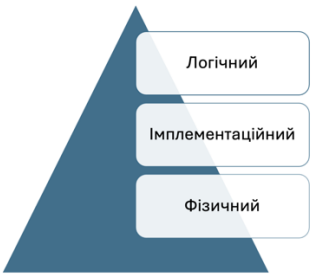


Рис.7. Рівні опису даних

1. Логічний (абстрактний) рівень

Це найвищий рівень, погляд "з висоти пташиного польоту", рівень ідей опис структури даних з точки зору користувача. Цей рівень відповідає на питання, *що* ця структура робить, *які* операції вона підтримує, та *які* властивості має її поведінка. На цьому рівні ми повністю ігноруємо, *як* це реалізовано всередині. Це є абстрактний тип даних (АТД). Це, по суті, "контракт" або специфікація. АТД визначає набір операцій та їхню очікувану поведінку. нариклад, АТД "Черга" (Queue). На логічному рівні ми визначаємо її як колекцію елементів, що працює за принципом "перший увійшов — перший вийшов" (FIFO).

- **Операції:**
  - enqueue(element): додати елемент у кінець черги.
  - dequeue(): видалити та повернути елемент з початку черги.
  - is_empty(): перевірити, чи черга порожня

Нам абсолютно неважливо, як саме реалізовані ці операції, головне — вони дотримуються "контракту" FIFO. Це дає нам, як розробникам, величезну гнучкість: ми можемо почати використовувати "Чергу" у своїй програмі, а потім, за потреби, замінити її внутрішню реалізацію з повільної на швидку, не змінюючи жодного рядка коду, який її використовує. Інший приклад — АТД "Стек" (Stack), який працює за принципом "останній увійшов — перший вийшов" (LIFO).

Коли ми говоримо про "шар нейронної мережі" в бібліотеці Keras, ми часто мислимо на цьому рівні. Ми знаємо, що Dense(512) приймає тензор на вхід, виконує обчислення і видає тензор розміром 512 на вихід. Ми не замислюємося про конкретні цикли чи функції, що реалізують множення матриць "під капотом".

2. Імплементаційний (структурний) рівень

Це рівень реалізації, міст між абстрактною ідеєю та фізичною пам'яттю, опис того, *як* саме логічна структура та її операції реалізовані за допомогою базових можливостей

мови програмування. Тут ми обираємо конкретні інструменти для втілення нашої ідеї. Наприклад, наш АТД "Черга" можна реалізувати кількома способами, і цей вибір має наслідки:

- На основі масиву (списку Python) ми просто додаємо елементи в кінець списку, а видаляємо з початку. *Недолік:* видалення з початку списку в Python — це повільна операція (O(n)), бо всі наступні елементи треба фізично "зсунути" в пам'яті вліво. Для великої черги це буде дуже неефективно.
- На основі зв'язного списку ми створюємо ланцюжок вузлів, де кожен вузол посилається на наступний. Додавання та видалення з обох кінців є дуже швидкими операціями (O(1)), оскільки потрібно лише змінити кілька посилань.

Як це пов’язано зі світом ШІ? Тут приймаються критично важливі для продуктивності рішення. Чи будемо ми зберігати вектор ознак у звичайному списку Python чи у масиві бібліотеки NumPy? Для математичних операцій NumPy array на порядки швидший, бо його імплементація значно ближча до фізичного рівня, використовує скомпільований код на C і оптимізована для цього.

3. Фізичний рівень

Це "залізо", найнижчий рівень представлення, світ бітів та байтів, опис того, як структури даних фізично розташовуються в пам'яті комп'ютера, як операційна система виділяє під них пам'ять, як працює кеш процесора. Наприклад, масив цілих чисел [10, 20] буде представлений у пам'яті як неперервний (contiguous) блок байтів. Якщо одне число займає 4 байти, то 10 буде лежати за адресою 0x1000, а 20 — одразу за ним, за адресою 0x1004. На противагу, зв'язний список зберігається в пам'яті фрагментовано: його вузли можуть бути розкидані по різних адресах, а зв'язок між ними підтримується за допомогою вказівників (адрес).

Програмісти ШІ рідко працюють на цьому рівні безпосередньо, але його розуміння пояснює, чому одні структури швидші за інші. Неперервне зберігання даних у масивах NumPy забезпечує "дружність до кешу" (cache locality). Процесору (особливо GPU) значно швидше завантажити один великий суцільний блок даних у свій швидкий кеш, ніж стрибати по різних адресах пам'яті, збираючи дані по шматочках. Це дозволяє обробляти дані паралельно за допомогою спеціальних векторних інструкцій (SIMD) і є однією з головних причин приголомшливої швидкості сучасних обчислень у глибокому навчанні.

Отже, розуміння цих трьох рівнів дозволяє керувати складністю. Ми можемо проектувати систему на логічному рівні, обирати найбільш ефективну реалізацію на імплементації, і при цьому розуміти, чому наш вибір буде швидким чи повільним завдяки знанням про фізичний рівень. Це і є шлях від абстрактної ідеї до оптимізованого коду, шлях, який перетворює програміста на інженера-архітектора.

1.8. Як комп'ютер "бачить" дані: Структури даних в пам'яті

Коли ми пишемо код, ми оперуємо зручними абстракціями: списками, словниками, об'єктами. Це дозволяє нам фокусуватися на логіці задачі, не замислюючись про "залізо". Але для комп'ютера вся оперативна пам'ять (RAM) — це просто один

величезний, довгий ряд пронумерованих комірок, кожна з яких може зберігати байт інформації. Адреса комірки — це її унікальний номер.

Те, як саме наші зручні абстракції розташовуються в цих фізичних комірках, докорінно впливає на продуктивність. Розуміння цих процесів "під капотом" і є тим, що відрізняє просто кодера від інженера-архітектора. Існує два фундаментальні підходи до організації даних у пам'яті: неперервний та зв'язний.

1. Статичне та динамічне виділення пам'яті: Стек проти Купи

Перш ніж говорити про організацію, варто згадати про два "відділи" пам'яті, де програма може "попросити" місце для своїх даних:

- Статичне виділення пам'яті (на стеку): Пам'ять для змінних виділяється під час компіляції програми, оскільки їхній розмір та кількість відомі заздалегідь. Цей "відділ" пам'яті називається стеком (Stack). Він працює за принципом "останній увійшов — перший вийшов" і є надзвичайно швидким, бо виділення та звільнення пам'яті — це просто зсув одного вказівника (stack pointer). Однак він дуже обмежений за розміром і негнучкий. У мовах високого рівня, як Python, ми рідко керуємо цим напряму, але саме на стеку зазвичай створюються прості змінні (числа, логічні значення) та посилання на об'єкти.
- Динамічне виділення пам'яті (в купі): Пам'ять виділяється під час виконання програми, коли виникає така потреба. Цей значно більший "відділ" пам'яті називається купою (Heap). Це дозволяє створювати структури даних, розмір яких невідомий заздалегідь (наприклад, список, до якого користувач може додавати елементи). Цей спосіб набагато гнучкіший, але й повільніший, оскільки вимагає більш складного механізму пошуку вільного блоку пам'яті та може призводити до її фрагментації. Більшість складних структур даних, з якими ми працюємо, розміщуються саме в купі. Навіть якщо посилання на список живе на швидкому стеку, сам масив елементів знаходиться у великій, гнучкій, але повільнішій купі.

2. Неперервне зберігання (Contiguous Storage): Принцип "Товарного потяга"

Це найпростіший і найпоширеніший спосіб організації даних у купі. Усі елементи структури даних зберігаються в пам'яті в одному суцільному, неперервному блоці, один за одним. За **аналогію можна взяти** товарний потяг. Усі вагони зчеплені разом і рухаються як єдине ціле. Щоб потрапити з п'ятого вагона в шостий, ви просто робите один крок.

Головний представник: Масив (Array).

- Переваги**:**
  - Швидкий доступ за індексом (O(1)): Щоб знайти елемент за індексом i, комп'ютеру не потрібно перебирати всі елементи. Він миттєво обчислює адресу за простою арифметичною формулою: адреса = адреса_початку + i * розмір_елемента.
  - «Дружність до кешу» (Cache Locality): Це найважливіша перевага. Уявіть, що ваша оперативна пам'ять — це великий склад, а кеш процесора — це ваш робочий стіл. Щоб приготувати страву, значно ефективніше один раз сходити на склад і принести на стіл усі потрібні інгредієнти, ніж бігати за кожним окремо. Так само і процесор: оскільки в масиві дані лежать поруч, він завантажує одразу великий блок у свій швидкий кеш. Завдяки цьому послідовна обробка даних (наприклад, у циклі for) відбувається надзвичайно швидко, бо всі потрібні «інгредієнти» вже під рукою.
- Недоліки:

- Складність зміни розміру: У низькорівневих мовах масиви мають фіксований розмір. У Python стандартний list є динамічним масивом: коли в ньому закінчується місце, Python створює новий, більший масив, і копіює туди всі старі елементи. Хоча це робить списки зручними, ця операція копіювання може іноді викликати помітні затримки.
- Повільні вставки та видалення (O(n)): Щоб вставити елемент у середину масиву на мільйон елементів, потрібно "розсунути" всі наступні елементи, перемістивши їх у пам'яті. Це може вимагати сотень тисяч операцій копіювання, що є дуже дорогою операцією.

- Роль в ШІ: Критично важлива. Тензори (основа TensorFlow та PyTorch) та масиви NumPy використовують саме неперервне зберігання. Саме це дозволяє GPU виконувати тисячі математичних операцій (як-от множення матриць) паралельно над величезними блоками даних, що є ключем до швидкості сучасного глибокого навчання.

3. Зв'язне зберігання (Linked Storage)

Принцип "Ланцюга автомобілів" - це альтернативний підхід, що вирішує головну проблему неперервного зберігання — повільні модифікації. Елементи структури (вузли) можуть бути розкидані по всій пам'яті. Кожен вузол містить не тільки самі дані, а й вказівник (посилання) — адресу наступного вузла в послідовності. Уявіть квест, де кожна наступна підказка захована в новому місці по всьому місту. Щоб знайти п'яту підказку, ви зобов'язані спочатку знайти першу, яка приведе вас до другої, потім до третьої і так далі. Ви не можете «перестрибнути» одразу до п'ятої, навіть якщо знаєте її номер. Так само працює і доступ до елементів у зв'язному списку.

Головний представник: Зв'язний список (Linked List).
Переваги:

- Динамічний розмір: Легко додавати нові елементи без необхідності переміщувати старі, оскільки ми просто знаходимо вільне місце в пам'яті і оновлюємо посилання.
- Швидкі вставки та видалення (O(1)): Щоб вставити новий елемент між двома існуючими, потрібно лише змінити два посилання. Це значно швидше, ніж "розсувати" масив.

Недоліки:

- Повільний доступ за індексом (O(n)): Щоб знайти 5-й елемент, потрібно послідовно пройти від першого елемента через усі проміжні. Немає способу "перестрибнути" одразу до потрібного.
- Більше використання пам'яті: Кожен вузол, окрім даних, має зберігати ще й адресу наступного елемента (або навіть двох, у випадку двозв'язного списку), що створює додаткові накладні витрати.
- "Недружність до кешу": Це головний недолік. Оскільки дані розкидані по пам'яті, процесор змушений постійно звертатися до повільної RAM, що призводить до "промахів кешу" (cache misses) і значно сповільнює обробку.

У ШІ використовуються рідше для числових обчислень, але можуть бути корисними для створення гнучких структур, наприклад, у генетичних алгоритмах для представлення хромосом змінної довжини або для реалізації черг та стеків у деяких алгоритмах обходу графів (наприклад, при реалізації BFS).

***Таблиця 3**. Структури даних у пам'яті комп'ютера*

Характеристика	**Неперервне зберігання (Масив)**	**Зв'язне зберігання (Зв'язний список)**
Розташування	Суцільний блок пам'яті	Розкидано по пам'яті
Доступ за індексом	Дуже швидкий (O(1))	Повільний (O(n))
Вставка/Видалення	Повільні (O(n))	Дуже швидкі (O(1))
Розмір	Зазвичай фіксований	Динамічний
Ефективність кешу	Висока	Низька

Отже, розуміння того, як дані розташовуються в пам'яті, дозволяє зробити усвідомлений вибір, який є одним із фундаментальних інженерних компромісів у розробці програмного забезпечення. Для інтенсивних математичних обчислень над великими, стабільними обсягами даних, як у ШІ, неперервне зберігання (масиви, тензори) є беззаперечним переможцем завдяки швидкості доступу та ефективності кешування. Для задач, де структура даних постійно і непередбачувано змінюється, а швидкість доступу за індексом не є критичною, зв'язне зберігання може бути кращим вибором.

1.9. Карта світу даних: Класифікація структур даних

Щоб орієнтуватися у величезному світі структур даних, програмісти використовують різні системи класифікації. Це схоже на те, як біологи класифікують живі організми: розуміння, до якого "царства" чи "типу" належить структура, дозволяє одразу зробити висновки про її ключові властивості, поведінку, сильні та слабкі сторони. Зазвичай структури даних класифікують за двома основними критеріями:

- За логічною організацією, як дані пов'язані між собою з точки зору програміста, тобто яка їхня концептуальна архітектура.
- За фізичним розташуванням, як дані насправді зберігаються в пам'яті комп'ютера, що безпосередньо впливає на їхню продуктивність.

Розглянемо кожну з них детальніше.

Класифікація 1: За логічною організацією (у програмах користувача)

Це найважливіша класифікація, оскільки вона визначає, як ми *мислимо* про дані та взаємодіємо з ними. Вона описує абстрактну модель, яку ми будуємо в голові, перш ніж написати перший рядок коду.

A. Лінійні структури даних

У цих структурах елементи організовані послідовно, один за одним, утворюючи ланцюжок. Кожен елемент (крім першого та останнього) має не більше одного

попередника та одного наступника, аналогічно як перлини на нитці, вагони в потязі, люди в черзі. Існує чіткий початок і кінець, а також поняття "наступного" та "попереднього" елемента. Це найпростіший спосіб упорядкування даних. Наприклад:

- масив (Array) - найпростіша лінійна структура, послідовність елементів одного типу, що зберігаються в пам'яті поруч. Її головна перевага — миттєвий доступ до будь-якого елемента за його індексом.
- Стек (Stack): Працює за принципом "Останній увійшов — перший вийшов" (LIFO - Last-In, First-Out). Це масив або список з обмеженим інтерфейсом: додавати (push) та видаляти (pop) елементи можна лише з одного кінця ("вершини" стека). Аналогія: стопка тарілок або історія переглядів у браузері (кнопка "Назад" дістає останню відвідану сторінку).
- Черга (Queue), яка працює за принципом "Перший увійшов — перший вийшов" (FIFO - First-In, First-Out). Елементи додаються в один кінець, а видаляються з іншого. Аналогія: черга в магазині або черга завдань на друк.
- Зв'язний список (Linked List) - послідовність елементів (вузлів), де кожен знає адресу наступного. На відміну від масиву, елементи не обов'язково лежать поруч у пам'яті, що дає гнучкість при змінах.

Перераховані типи даних ідеально підходять для роботи з послідовними даними: текст (послідовність слів), аудіосигнал (послідовність семплів), часові ряди (послідовність вимірювань). Стеки використовуються в алгоритмах пошуку в глибину (DFS), а черги — в алгоритмах пошуку в ширину (BFS), які є основою для обходу дерев та графів.

Б. Нелінійні структури даних

На відміну від лінійних структур, де кожен елемент має лише один шлях 'вперед' і один 'назад', нелінійні структури дозволяють даним мати багаторівневі та багатовимірні зв'язки. Елемент тут може бути батьком для багатьох нащадків або сусідом для багатьох інших вершин. Один елемент може бути пов'язаний з багатьма іншими, що дозволяє моделювати складні системи реального світу подібно до родинного дерева, карти метро, соціальної мережі. Відносини між даними є більш складними, ніж просто "наступний/попередній". Існують поняття "батько-нащадок", "предок-наступник", "сусід".

Основні типи:

1. Дерева (Trees): Це ієрархічна структура, що складається з вузлів, з'єднаних ребрами. Має один кореневий елемент, від якого відходять гілки. Кожен елемент (крім кореня) має одного "батька" і може мати багато "нащадків". Важливо: у деревах немає циклів, що гарантує чітку ієрархію і єдиний шлях від кореня до будь-якого вузла.
   - Приклади: Файлова система на комп'ютері, DOM-структура веб-сторінки, генеалогічне дерево.
   - Роль в ШІ: Дерева рішень — це один із найпопулярніших алгоритмів класичного машинного навчання, де кожен внутрішній вузол представляє перевірку ознаки, а кожен листовий вузол — кінцеве рішення. У задачах обробки природної мови використовуються синтаксичні дерева для аналізу структури речень.
2. Графи (Graphs): Найбільш загальна та гнучка структура даних. Складається з набору вершин (вузлів) та ребер (зв'язків), що їх з'єднують. На відміну від дерев, у графах можуть бути цикли, а вершина може не мати "батьків" або мати їх декілька. Будь-яка вершина може бути з'єднана з будь-якою іншою.
   - Приклади: Карта доріг, мережа Інтернет, зв'язки між людьми у Facebook.

- Роль в ШІ: Критично важлива. Самі нейронні мережі за своєю суттю є складними, спрямованими, зваженими графами, де нейрони — це вершини, а синапси — ребра з вагами. Графи знань використовуються для зберігання структурованої інформації про світ (напр., "Київ — столиця України"). Аналіз соціальних мереж, рекомендаційні системи, пошук оптимальних шляхів у робототехніці — все це задачі на графах.

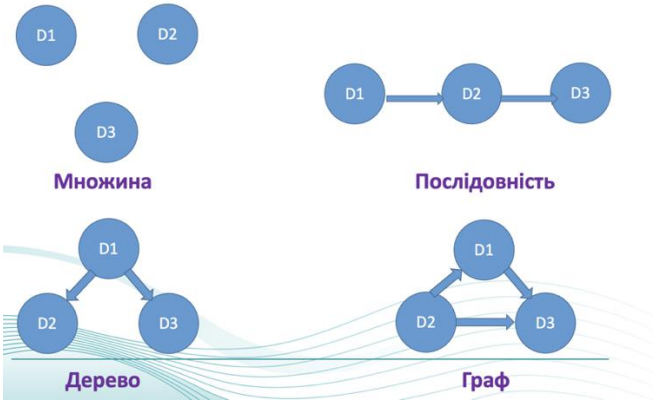


Рис.8. Приклади подання структур даних

Класифікація 2: За фізичним розташуванням (у пам'яті комп'ютера)

Ця класифікація описує, як структури даних фізично розташовуються в оперативній пам'яті, що є ключовим фактором їхньої продуктивності.

А. Структури неперервного зберігання (Contiguous)

В структурах даних неперервного зберігання всі елементи зберігаються в одному суцільному блоці пам'яті, як солдати в строю. Представником такого типу даних є масиви, які відрізняються швидким доступом до будь-якого елемента за його індексом (O(1)), але повільними операціями вставки та видалення елементів, бо це вимагає "зсуву" всіх наступних елементів. Це ідеальний вибір для даних, які рідко змінюються, але часто читаються.

Б. Структури зв'язного зберігання (Linked)

В структурах зв’язного зберігання елементи (вузли) можуть бути розкидані по всій пам'яті, як острови в архіпелазі. Зв'язок між ними підтримується за допомогою вказівників (посилань), які є "мостами" між островами. Представниками таких структур є зв'язний список, дерева, графи. Вони мають швидкі операції вставки та видалення елементів, оскільки потрібно лише перебудувати кілька "мостів", а не рухати цілі "острови". Проте доступ до елемента за індексом є повільним,, оскільки потрібно послідовно пройти по ланцюжку. Це гарний вибір для даних, що часто змінюються.

Таблиця 4. *Класифікація структур даних*

Структура даних	Логічна організація	Типове фізичне зберігання	Головна перевага	Основний недолік
Масив	Лінійна	Неперервне	Швидкий доступ за індексом	Повільні вставки/видалення
Стек	Лінійна	Неперервне (на базі масиву)	Простота, швидкість операцій	Обмежений функціонал
Черга	Лінійна	Зв'язне або неперервне (кільцевий буфер)	Дотримання порядку FIFO	Обмежений функціонал
Зв'язний список	Лінійна	Зв'язне	Швидкі вставки/видалення	Повільний доступ за індексом
Дерево	Нелінійна	Зв'язне	Ефективний пошук, ієрархія	Складність балансування
Граф	Нелінійна	Зв'язне (списки суміжності)	Моделювання складних зв'язків	Висока складність алгоритмів

Отже, розуміння цих класифікацій дозволяє зробити усвідомлений вибір. Коли ми бачимо задачу, ми спершу визначаємо, яка логічна структура найкраще моделює наші дані (послідовність, ієрархія чи мережа?). Потім ми думаємо, яка фізична реалізація цієї структури буде найефективнішою з огляду на операції, які ми плануємо виконувати найчастіше (часто читати, чи часто змінювати?). Саме в цьому і полягає інженерне мистецтво вибору правильної структури даних, яке перетворює абстрактну ідею на швидкий та надійний код.

1.10. Будівельні блоки інформації: Основні види простих і складних типів даних

Будь-яка інформація, з якою працює комп'ютер — від вашого імені до складної моделі штучного інтелекту з мільярдами параметрів — побудована з дуже простих, фундаментальних "атомів" даних. Розуміння цих базових типів та способів їх поєднання у складні структури є ключем до того, щоб навчити машину "розуміти" та обробляти інформацію про реальний світ. У цьому розділі ми розглянемо ці фундаментальні будівельні блоки, від найпростіших "атомів" (примітивних типів) до складних "молекул" (композитних типів), і побачимо, яку критичну роль кожен з них відіграє в архітектурі сучасного штучного інтелекту.

Частина 1: Прості (примітивні) типи даних

Це "атоми" світу даних. Вони є неподільними з точки зору мови програмування і є основою для всього іншого. Вони безпосередньо відповідають тому, як процесор може маніпулювати даними на найнижчому рівні.

1. Цілі числа (Integer)

Що це? Прості цілі числа без дробової частини (напр., -5, 0, 25, 2024). У більшості мов вони мають обмежений діапазон (напр., 32-бітні або 64-бітні), але Python елегантно обходить це, дозволяючи працювати з довільно великими цілими числами, автоматично виділяючи більше пам'яті за потреби.

Роль в ШІ:

- Лічильники. Кількість епох навчання, номер поточної ітерації в циклі оптимізації. Це "пульс" процесу навчання.
- Індекси. Номер елемента у векторі ознак або номер пікселя в зображенні. Це фундаментальний механізм доступу до даних у масивах, що лежить в основі всіх матричних операцій.
- Категоріальні мітки. У задачах класифікації ми часто кодуємо класи числами: 0 — "кіт", 1 — "собака", 2 — "птах". Це дозволяє перетворити нечислові дані на формат, зрозумілий для математичних моделей.
- Кількість нейронів. Dense(512) — тут 512 є цілим числом, що задає архітектуру шару. Від цього числа безпосередньо залежить "ємність" моделі, її здатність запам'ятовувати складні залежності.

2. Числа з плаваючою комою (Float)

Що це? Дійсні числа, що можуть мати дробову частину (напр., -3.14, 0.001, 99.9). Вони існують у різних форматах точності (напр., float32 - одинарна точність, float64 - подвійна).

Роль в ШІ. Це найважливіший тип даних для сучасного ШІ.

- Ваги та зміщення (Weights & Biases): Абсолютно всі параметри, які "навчаються" в нейронній мережі, є числами типу float. Це мільярди маленьких "регуляторів", які модель тонко налаштовує під час навчання. Вибір точності (напр., float16 замість float32) є важливим компромісом: менша точність дозволяє вдвічі зменшити обсяг пам'яті та прискорити обчислення на GPU, але може вплинути на якість моделі.
- Ймовірності: Результат роботи класифікатора — це зазвичай ймовірність приналежності до класу, що є числом від 0.0 до 1.0 (напр., "на 98.7% впевнений, що це кіт").
- Нормалізовані дані: Вхідні дані (напр., вік, дохід) майже завжди нормалізують, приводячи до діапазону [0, 1] або [-1, 1]. Це робиться для того, щоб ознаки з великими значеннями (як дохід) не "домінували" над ознаками з малими значеннями (як вік) під час навчання.
- Векторні представлення (Embeddings): Кожне слово або зображення у сучасних моделях представляється у вигляді вектора з сотень чисел типу float. Цей вектор — не просто набір чисел, а координата в багатовимірному "просторі значень". Слова зі схожим значенням (напр., "король" та "королева") будуть знаходитись близько одне до одного в цьому просторі.

3. Логічний (булів) тип (Boolean)

Що це? Тип, що може мати лише два значення: True (істина) або False (хиба). Це цифровий еквівалент відповіді "так" або "ні".

Роль в ШІ:

- Бінарна класифікація: Результат роботи спам-фільтру (is_spam: True) або медичного тесту (has_disease: False).

- Керування логікою: Використовується в умовах для керування процесом навчання. Наприклад, критерій ранньої зупинки (early stopping): if validation_loss_not_improving_for_5_epochs: stop_training = True.
- Ознаки (Features): Деякі ознаки за своєю природою є бінарними (напр., is_married: True, has_higher_education: False).
- Маски (Masks): У обробці тексту булеві масиви використовуються, щоб вказати моделі, на які частини речення (напр., реальні слова) треба звертати увагу, а які ігнорувати (напр., спеціальні символи-заповнювачі).

4. Рядки (String)

Що це? Послідовність символів, взята в лапки (напр., "Привіт, світе!", "cat_image_01.jpg").

Роль в ШІ:

- Вхідні дані для NLP: Будь-яка задача обробки природної мови (Natural Language Processing) починається з рядків. Токенізація — процес розбиття рядка на менші частини (слова або частини слів) — є першим кроком перетворення тексту на щось, з чим може працювати модель.
- Імена файлів та шляхи: Для завантаження датасетів, збереження моделей та логування результатів.
- Категоріальні ознаки: Значення ознак, як-от "місто": "Київ" або "професія": "інженер". Модель не може працювати з ними напряму, тому вони перетворюються на числові представлення.

Частина 2: Складні (композитні) типи даних

Це "молекули", які ми будуємо, комбінуючи прості типи. Вони дозволяють нам моделювати складні об'єкти реального світу, зберігаючи при цьому їхню структуру та взаємозв'язки.

1. Масив / Список (Array / List)

Що це? Впорядкована, змінювана колекція елементів.

Роль в ШІ. Фундамент усього.

- Вектор ознак. Один об'єкт (напр., клієнт) описується як список його числових характеристик: [вік, дохід, кількість_покупок].
- Зображення. Представляється як двовимірний масив (матриця) пікселів.
- Текст. Після токенізації речення перетворюється на список індексів слів згідно зі словником: [101, 256, 3005, 102].
- Тензор. Узагальнення масиву на будь-яку кількість вимірів. Це "рідна мова" бібліотек TensorFlow та PyTorch. Наприклад, 4D-тензор може представляти собою цілу партію (batch) зображень: (кількість_зображень, висота, ширина, кількість_каналів). Вся робота в глибокому навчанні — це операції над тензорами.

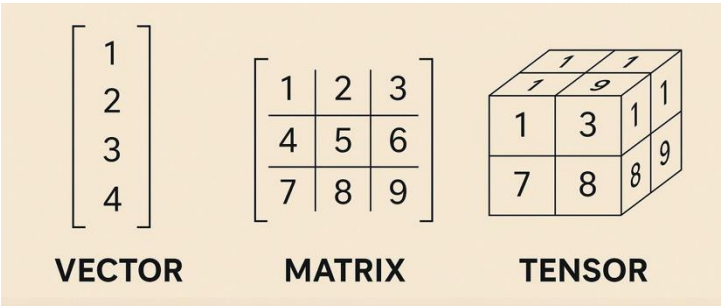


Рис.9. Ілюстрація різниці між вектором, матрицею та тензором

2. Словник / Хеш-таблиця (Dictionary / Hash Table)

- Що це? Невпорядкована колекція пар "ключ-значення", що забезпечує надзвичайно швидкий доступ до значення за його унікальним ключем. Ця швидкість досягається завдяки хешуванню — процесу, де ключ перетворюється на індекс у внутрішньому масиві.
- Роль в ШІ:
  - Конфігурація експериментів: Зберігання гіперпараметрів моделі: config = {"learning_rate": 0.001, "epochs": 100, "optimizer": "Adam"}. Це дозволяє гнучко керувати експериментами.
  - Словник (Vocabulary) в NLP: Створення відповідності між словом та його унікальним числовим індексом: {"слово": 1, "було": 2, ...}.
  - Представлення даних: Формат JSON, який є по суті словником, широко використовується для обміну даними через API. Наприклад, AI-сервіс може повернути результат у вигляді: {"label": "кіт", "confidence": 0.987}.

3. Кортеж (Tuple)

- Що це? Впорядкована, але незмінна (immutable) колекція елементів. Після створення кортеж не можна змінити.
- Роль в ШІ: Використовується там, де дані не повинні змінюватися, що гарантує стабільність та передбачуваність.
  - Форма (Shape) тензора: Розміри тензора, напр., (64, 224, 224, 3) (64 зображення розміром 224x224 з 3 колірними каналами), завжди представляються у вигляді кортежу, бо архітектура моделі залежить від незмінної форми даних. Випадкова зміна цього значення призвела б до краху програми.
  - Ключі в словнику: Кортеж, на відміну від списку, може бути ключем у словнику, що дозволяє створювати складні ключі (напр., {(x, y): "значення"}).

4. Множина (Set)

- Що це? Невпорядкована колекція унікальних елементів.
- Роль в ШІ:
  - Побудова словника: Найшвидший спосіб зібрати всі унікальні слова з великого тексту перед тим, як призначати їм індекси.
  - Аналіз даних: Швидко знайти всі унікальні категорії в колонці датасету (напр., усі унікальні міста, з яких приїхали клієнти).
  - Оцінка моделей: При порівнянні результатів роботи двох моделей можна використовувати операції над множинами (перетин, об'єднання) для аналізу того, наскільки їхні передбачення збігаються.
  - Перевірка наявності: Множини забезпечують дуже швидку (O(1)) перевірку, чи міститься елемент у колекції.

Отже, шлях від бітів та байтів до інтелектуальних систем лежить через правильну організацію даних. Розуміння того, як прості типи (числа, рядки) поєднуються у складні структури (масиви, словники), і як кожна з цих структур використовується для представлення різних аспектів реального світу, є першим і найважливішим кроком до створення ефективних та потужних моделей штучного інтелекту. Це не просто інструменти, а мова, якою ми описуємо світ для машини.

Контрольні запитання до розділу

1. Сутність алгоритму
Чому недостатньо просто сказати, що алгоритм — це "набір інструкцій"? Які ключові властивості (наприклад, скінченність, визначеність, ефективність) перетворюють звичайну інструкцію на повноцінний алгоритм, і чому кожна з цих властивостей є важливою?
2. Абстракція та реалізація
Поясніть своїми словами різницю між поняттями "алгоритм" та "програма". Чи може один і той самий алгоритм мати багато різних програмних реалізацій? Наведіть приклад.
3. Ціна ефективності
Навіщо нам вивчати ефективність алгоритмів (О-нотацію), якщо сучасні комп'ютери є надзвичайно швидкими? Уявіть, що у вас є два алгоритми для вирішення однієї задачі: один працює за час $O(n^2)$, а інший — за $O(n \log n)$. Як зміниться різниця в часі їхнього виконання, якщо розмір вхідних даних зросте з 1000 елементів до 1 000 000?
4. Алгоритми у повсякденному житті
Опишіть будь-який процес з вашого повсякденного життя, який не пов'язаний з комп'ютерами, у вигляді алгоритму (наприклад, рецепт приготування страви, інструкція зі збирання меблів). Чи задовольняє ваша інструкція всім властивостям алгоритму?
5. Еволюція алгоритмів в епоху ШІ
Чим принципово відрізняється класичний детермінований алгоритм (наприклад, сортування) від алгоритму машинного навчання (наприклад, класифікації зображень)? Чи можна сказати, що модель ШІ — це алгоритм, який "пише сам себе" на основі даних?
6. Компроміс при виборі
Уявіть, що ви розробляєте систему для навігатора. У вас є два алгоритми пошуку шляху: один дуже швидкий, але не завжди знаходить найкоротший маршрут, а інший завжди знаходить оптимальний шлях, але працює значно повільніше. Які фактори ви б враховували, обираючи між ними для реального продукту?
7. Соціальний вимір
Алгоритми рекомендацій у соціальних мережах та на стрімінгових сервісах впливають на те, яку інформацію ми споживаємо. Поміркуйте, які можуть бути позитивні та негативні соціальні наслідки роботи цих алгоритмів.
8. "Чорна скринька"
Деякі складні алгоритми ШІ називають "чорними скриньками", оскільки навіть їхні розробники не завжди можуть точно пояснити, чому було прийнято те чи інше рішення. Чому "інтерпретованість" алгоритму може бути критично важливою у таких сферах, як медицина (постановка діагнозу) або фінанси (видача кредиту)?
9. Алгоритми та "залізо"
Як ви думаєте, як розвиток апаратного забезпечення (наприклад, поява потужних графічних процесорів - GPU) вплинув на те, які типи алгоритмів стали популярними та можливими для реалізації в останні десятиліття?
10. Особиста рефлексія
Який аспект вивчення алгоритмів видається вам найбільш складним або, навпаки, найбільш захоплюючим? Чи це розробка логіки, аналіз ефективності, чи розуміння їхнього практичного застосування? Чому?

Тести для самоконтролю

1. Що найкраще описує поняття "алгоритм"?
    a) Будь-яка інструкція для комп'ютера
    b) Фрагмент коду на мові Python
    c) Скінченна послідовність чітко визначених дій для вирішення задачі
    d) Математична формула
2. Яка з перелічених властивостей НЕ є обов'язковою для алгоритму?
    a) Скінченність (завершення за скінченну кількість кроків)
    b) Визначеність (кожен крок є чітким та однозначним)
    c) Ефективність (кожен крок має бути простим і виконуваним)
    d) Простота (алгоритм має бути легким для розуміння людиною)
3. У чому полягає основна різниця між алгоритмом та програмою?
    a) Алгоритм пишеться мовою програмування, а програма — ні.
    b) Алгоритм — це ідея або логіка вирішення задачі, а програма — це його конкретна реалізація певною мовою програмування.
    c) Алгоритми використовуються тільки в математиці, а програми — в комп'ютерах.
    d) Немає жодної різниці, це синоніми.
4. Що оцінює О-нотація (наприклад, O(n), O(log n))?
    a) Точний час виконання програми в секундах
    b) Кількість рядків коду в програмі
    c) Як зростає час виконання алгоритму зі збільшенням обсягу вхідних даних
    d) Кількість пам'яті, яку використовує алгоритм
5. Чим алгоритм машинного навчання принципово відрізняється від класичного алгоритму сортування?
    a) Він завжди працює швидше.
    b) Він не має чітко визначених кроків.
    c) Він "навчається" правилам на основі даних, а не слідує жорстко запрограмованій логіці.
    d) Він може працювати лише з числами.
6. Який тип алгоритму, швидше за все, лежить в основі роботи GPS-навігатора для прокладання маршруту?
    a) Алгоритм сортування
    b) Алгоритм пошуку шляху в графі
    c) Алгоритм стиснення даних
    d) Алгоритм шифрування
7. Коли ми говоримо про "чорну скриньку" в контексті ШІ, що мається на увазі?
    a) Алгоритм, який працює з секретними даними.
    b) Алгоритм, написаний на маловідомій мові програмування.
    c) Складний алгоритм (часто нейронна мережа), чий процес прийняття рішення важко або неможливо повністю зрозуміти та інтерпретувати.
    d) Алгоритм, який ще не був протестований.
8. Яка мета алгоритмів рекомендацій у соціальних мережах?
    a) Показувати всі пости від друзів у хронологічному порядку.
    b) Захищати дані користувача від зламів.
    c) Аналізувати поведінку користувача, щоб показувати йому найбільш релевантний та цікавий контент.
    d) Зменшувати час, який користувач проводить у додатку.

9. Чому вивчення ефективності алгоритмів (О-нотації) є важливим навіть при наявності потужних комп'ютерів?

a) Тому що різниця між ефективним (O(n log n)) та неефективним ($O(n^2)$) алгоритмом на великих даних стає колосальною і може вимірюватися годинами або навіть днями.

b) Це вимога для отримання сертифікату програміста.

c) Це допомагає писати коротший код.

d) Це не важливо, сучасні комп'ютери можуть впоратися з будь-яким алгоритмом.

10. "Скінченність" як властивість алгоритму означає, що:

a) Алгоритм може працювати лише зі скінченною кількістю даних.

b) Алгоритм повинен гарантовано завершити свою роботу за скінченну кількість кроків.

c) Кожен крок алгоритму має бути простим.

d) Алгоритм можна записати у вигляді скінченної кількості символів.

Завдання для самостійної роботи

Це завдання допоможе вам глибше зрозуміти фундаментальні концепції алгоритмів, їхні властивості та роль у сучасному світі.

Частина 1: Аналіз та рефлексія (Письмово)

Дайте розгорнуті відповіді на наступні питання.

1. Аналіз властивостей алгоритму. Опишіть своїми словами, що означають три ключові властивості алгоритму: скінченність, визначеність та ефективність. Чому інструкція, яка не задовольняє хоча б одній з цих властивостей (наприклад, не завершується або має двозначні кроки), не може вважатися повноцінним алгоритмом?
2. Оцінка ефективності (О-нотація). Уявіть, що у вас є два алгоритми для пошуку елемента у наборі даних з 1,000,000 записів.
   - Алгоритм А має складність O(n) (лінійну).
   - Алгоритм Б має складність O(log n) (логарифмічну). Поясніть, у скільки разів (приблизно) алгоритм Б буде швидшим за алгоритм А у найгіршому випадку. Чому ця різниця стає настільки колосальною при роботі з великими даними?
3. Проблема "чорної скриньки". Алгоритми ШІ, що використовуються для видачі кредитів або медичної діагностики, іноді працюють як "чорні скриньки". Поміркуйте, які потенційні ризики та етичні проблеми це створює для суспільства. Чому "інтерпретованість" (можливість пояснити рішення) алгоритму є такою важливою у цих сферах?

Частина 2: Практичне завдання (Розробка алгоритму)

Оберіть один із наведених нижче повсякденних процесів та опишіть його у вигляді покрокового алгоритму. Ваша мета — зробити інструкцію настільки чіткою, щоб її могла виконати людина, яка ніколи цього не робила.

Процеси на вибір:

- Приготування вашої улюбленої страви (наприклад, омлету або кави).
- Процес збирання на роботу/навчання вранці.

- Алгоритм прибирання кімнати.

Вимоги до опису:

1. Чітко визначте вхідні дані (наприклад, "інгредієнти: 2 яйця, 50 мл молока...") та вихідний результат ("готова страва: омлет").
2. Розпишіть процес як послідовність пронумерованих кроків.
3. Використовуйте умовні оператори (якщо... то...). Наприклад: "Якщо сковорідка достатньо гаряча, то вилити суміш".
4. Після написання перевірте, чи задовольняє ваш алгоритм ключовим властивостям (чи є він скінченним, визначеним та ефективним).

Частина 3: Дослідження та аналіз (Письмово)

Оберіть один із сучасних цифрових сервісів, якими ви користуєтесь щодня (наприклад, YouTube, TikTok, Netflix, Google Maps, Spotify).

1. Визначіть, який ключовий алгоритм лежить в основі його функціонування (наприклад, алгоритм рекомендацій, алгоритм пошуку шляху, алгоритм пошуку інформації).
2. Опишіть, що є вхідними даними для цього алгоритму (наприклад, історія переглядів, лайки, поточне місцезнаходження) і що є його результатом (наприклад, персоналізована стрічка відео, найкоротший маршрут).
3. Спробуйте зробити припущення, наскільки ефективним має бути цей алгоритм, враховуючи величезну кількість даних, які він обробляє.
4. Коротко поміркуйте, як цей алгоритм впливає на вашу поведінку та вибір.

Розділ 2: Python для ШІ - базовий інструментарій

Вступ

Після розгляду теоретичних основ алгоритмів та структур даних, наступним логічним кроком є перехід до їх практичної реалізації. Для втілення абстрактних концепцій у працюючий код необхідний відповідний інструментарій. У сучасній галузі штучного інтелекту та науки про дані домінуючою мовою програмування, що виступає в ролі такого інструментарію, є Python.

Поширеність Python у сфері ШІ не є випадковою і обумовлена сукупністю трьох ключових факторів, що забезпечили йому конкурентну перевагу.

1. Простота синтаксису та висока читабельність: Однією з головних філософських засад Python є пріоритет читабельності коду. Синтаксис мови є мінімалістичним та інтуїтивно зрозумілим, що дозволяє розробнику зосередитись на логіці вирішення задачі, а не на боротьбі зі складними мовними конструкціями. Це значно прискорює процес прототипування та розробки.
2. Розвинена екосистема спеціалізованих бібліотек: Найбільшою перевагою Python є його потужна екосистема, що включає тисячі бібліотек з відкритим кодом, які стали промисловими стандартами:
    - NumPy: Фундаментальна бібліотека для наукових обчислень, що забезпечує підтримку багатовимірних масивів (тензорів) та високопродуктивних математичних операцій над ними.
    - Pandas: Надає високорівневі структури даних та інструменти для аналізу табличних та часових даних.
    - Matplotlib та Seaborn: Широкий інструментарій для створення статичних, анімованих та інтерактивних візуалізацій.
    - TensorFlow та PyTorch: Дві фундаментальні бібліотеки, на яких побудована переважна більшість сучасних архітектур глибокого навчання.
3. Низький поріг входження: Завдяки поєднанню простоти та потужної екосистеми, Python дозволяє фахівцям з різних галузей (математикам, фізикам, біологам, економістам), які не є професійними програмістами, швидко та ефективно застосовувати методи машинного навчання для вирішення своїх прикладних задач.

Отже, в рамках даної теми основна увага буде приділена опануванню базового інструментарію, що надається мовою Python безпосередньо. Студентам належить:

- Здійснити налаштування робочого середовища для розробки.
- Закріпити на практиці роботу зі змінними та основними типами даних.
- Навчитись використовувати вбудовані структури даних: списки, словники, кортежі та множини, як практичні реалізації раніше розглянутих абстрактних типів даних.
- Отримати навички організації коду за допомогою функцій та модулів з метою забезпечення його чистоти, модульності та можливості повторного використання.

Опанування цього базового інструментарію є необхідною умовою для подальшого вивчення та реалізації складних алгоритмів і є першим практичним кроком на шляху до розробки повноцінних інтелектуальних систем.

Історія Python розпочинається в кінці 1980-х років у Нідерландах. Її творцем є Гвідо ван Россум, співробітник Національного дослідницького інституту математики та інформатики (CWI).
Гвідо ван Россум народився 31 січня 1956 року в Гаазі, Нідерланди. Він відомий у всьому світі як творець мови програмування Python. Ось основні етапи його життя та кар'єри. З дитинства виявляв інтерес до науки та техніки. Навчався в Амстердамському університеті, де в 1982 році отримав ступінь магістра з математики та інформатики. Після закінчення університету почав працювати в Центрі математики та інформатики (CWI) в Амстердамі. Гвідо ван Россум працював над мовою програмування ABC, яка була розроблена для навчання програмуванню. Цей досвід вплинув на його подальшу роботу над Python. Він прагнув створити мову, яка була б легкою для читання, зрозумілою та потужною. Наприкінці 1980-х років почав роботу над проєктом Python, який був випущений у 1991 році. У лютому 1991 року Гвідо ван Россум випустив першу версію Python (0.9.0). Назва "Python" походить від назви комедійного телешоу "Літаючий цирк Монті Пайтона", яке подобалося Гвідо. Python швидко набув популярності завдяки своєму чистому синтаксису, великій стандартній бібліотеці та підтримці різних парадигм програмування. З'явилася велика спільнота розробників, які внесли значний внесок у розвиток мови. 2008 року вишла версія Python 3.0, яка містила ряд змін, несумісних з попередніми версіями.
Python є однією з найпопулярніших мов програмування у світі. Він використовується в різних галузях, включаючи веб-розробку, аналіз даних, штучний інтелект та наукові обчислення. Python продовжує активно розвиватися, з регулярними випусками нових версій.
Python — це інтерпретована мова, тобто код виконується построково, без попередньої компіляції. Python — це мова з відкритим вихідним кодом, що означає, що вона є безкоштовною для використання та розповсюдження.
Python швидко став популярним завдяки своєму чистому синтаксису, великій стандартній бібліотеці та підтримці різних парадигм програмування.

2.1. Перший крок до коду: Встановлення та робота в середовищі IDLE

Перш ніж ми зможемо писати програми на Python, нам потрібно підготувати наше робоче місце — встановити саму мову програмування та середовище розробки (IDE), в якому ми будемо писати та запускати код.

Для початківців найкращим вибором є **IDLE** (Integrated Development and Learning Environment) — просте та легке середовище, яке встановлюється разом з офіційним дистрибутивом Python. Воно не має надлишкового функціоналу, що дозволяє зосередитись на вивченні самої мови.

Етап 1: Встановлення Python та IDLE

IDLE не потрібно встановлювати окремо — він є частиною офіційного інсталятора Python.

1. Перейдіть на офіційний сайт. Відкрийте ваш веб-браузер та перейдіть на офіційну сторінку завантаження Python: python.org/downloads/.

2. Завантажте інсталятор. Сайт автоматично визначить вашу операційну систему (Windows, macOS) і запропонує завантажити останню стабільну версію. Натисніть на кнопку "Download Python".
3. Запустіть інсталятор (Дуже важливий крок для Windows):
   - Відкрийте завантажений файл.
   - ОБОВ'ЯЗКОВО поставте галочку навпроти "Add Python to PATH" внизу вікна інсталятора. Цей крок дозволить вам запускати Python з будь-якого місця у командному рядку, що буде необхідно в майбутньому.
   - Натисніть "Install Now", щоб розпочати стандартне встановлення.
4. Перевірка встановлення.
   - Після завершення встановлення відкрийте меню "Пуск" (у Windows) або пошук (у macOS).
   - Введіть IDLE і запустіть знайдену програму. Якщо відкрилося вікно з написом "Python ... Shell", все встановлено правильно.

Уявіть, що IDLE — це ваш робочий зошит з програмування. Він має два режими: «пісочниця» (інтерактивна оболонка) для експериментів та «чистовик» (редактор файлів) для написання готових робіт.

Режим 1: Інтерактивна оболонка (Shell)

Це вікно, яке ви бачите одразу після запуску IDLE. Воно легко впізнається за запрошенням >>>. За стрілками ви впізнаєте інтерактивний калькулятор для Python. Ви вводите одну команду, натискаєте Enter, і одразу ж бачите результат її виконання.

В інтерактивній оболонці ви можете перевірити, як працює певна функція, або виконати прості математичні обчислення. Вона ідеально підходить для того, щоб "погратися" з синтаксисом мови та миттєво побачити результат, також використовується для налагодження (Debugging), можна швидко перевірити значення змінної.

Приклад роботи в оболонці:

```
>>> 2 + 2
4
>>> message = "Привіт, AI-First!"
>>> print(message)
Привіт, AI-First!
>>> message.upper()
'ПРИВІТ, AI-FIRST!'
```

Оболонка не призначена для написання програм, що складаються з багатьох рядків, оскільки код не зберігається.

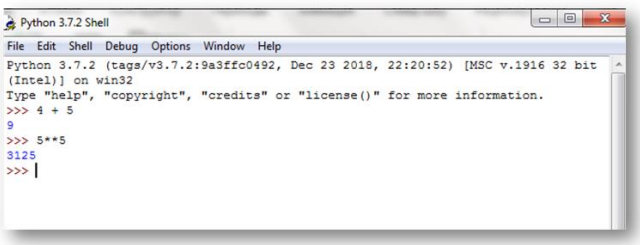


Рис. 10. Інтерактивна оболонка IDLE

Режим 2: Редактор файлів (File Editor)

Це режим, в якому ви будете писати ваші програми та проекти.

- Як відкрити? В інтерактивній оболонці натисніть **File -> New File** (або Ctrl+N). Відкриється нове, порожнє вікно без запрошення >>>.
- Що це? Це простий текстовий редактор, спеціалізований для написання коду на Python (він підсвічує синтаксис, допомагає з відступами тощо).
- Для чого використовується? Для написання повноцінних програм, що складаються з багатьох рядків, функцій та класів.
- **Як працювати:**
  1. Написання коду. Введіть вашу програму в новому вікні редактора. # Це моя перша програма name = input("Як вас звати? ") print(f"Привіт, {name}! Ласкаво просимо до курсу.")
  2. Збереження файлу. Натисніть **File -> Save** (або Ctrl+S). **Дуже важливо:** збережіть файл з розширенням .py (наприклад, my_first_program.py). Це стандартне розширення для файлів з кодом Python.
  3. Запуск програми. Натисніть **Run -> Run Module** (або клавішу **F5**).
- Результат. Після запуску програми результат її виконання (включаючи будь-який вивід print та запити input) з'явиться у вікні інтерактивної оболонки.

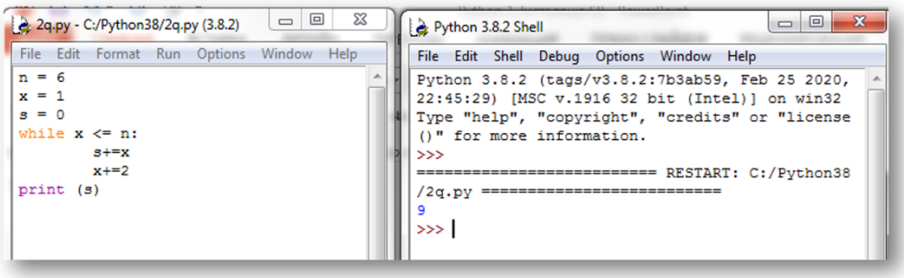


Рис. 11. Вікна IDLE поруч: вікно редактора з кодом та вікно оболонки, де видно результат запуску

Отже, IDLE — це чудовий стартовий майданчик. Навички, які ви тут отримаєте (написання коду у файлі, збереження, запуск), є універсальними. З часом ви, ймовірно, перейдете на більш потужні редактори, як-от VS Code, але фундаментальний процес залишиться тим самим. Тож опануйте ці два режими, і ви будете готові до виконання будь-яких практичних завдань у нашому курсі.

Приклад.
Перша програма
Ось проста програма на Python, яка виводить текст "Привіт, світ!" на екран:

```
Python
print("Привіт, світ!")
```

Як запустити цю програму:

1. Збережіть код:
   - Відкрийте текстовий редактор (наприклад, Блокнот, VS Code, Sublime Text).
   - Скопіюйте та вставте код.

    - Збережіть файл з розширенням .py, наприклад, hello.py.
2. Запустіть з командного рядка (терміналу):
    - Відкрийте командний рядок (Windows) або термінал (macOS/Linux).
    - Перейдіть до папки, де ви зберегли файл, за допомогою команди cd.
    - Введіть python hello.py і натисніть Enter.
3. Запустіть з IDLE:
    - Відкрийте IDLE.
    - Перейдіть до File -> Open, та відкрийте ваш файл hello.py.
    - Перейдіть до Run -> Run Module, або натисніть F5.

Пояснення коду:

- print() - це вбудована функція Python, яка виводить текст на екран.
- "Привіт, світ!" - це рядок (текст), який передається функції print().

Ця програма є класичним прикладом першої програми, яку пишуть при вивченні будь-якої мови програмування.

Другий варіант першої програми з використанням функцій.

```
def привітання(ім'я):
 """Ця функція виводить привітання з заданим ім'ям."""
 print(f"Привіт, {ім'я}!")

# Виклик функції
привітання("Іван")
привітання("Марія")
```

Пояснення:

1. **`def привітання(ім'я):`**
    - Це рядок визначає функцію з назвою `привітання`.
    - `ім'я` в дужках - це параметр, який функція приймає.
2. **`"""Ця функція виводить привітання з заданим ім'ям."""`:**
    - Це рядок документації (docstring), який описує, що робить функція.
3. **`print(f"Привіт, {ім'я}!")`:**
    - Це рядок коду, який виводить привітання, використовуючи параметр `ім'я`.
    - f-рядки (f-strings) дозволяють вставляти значення змінних безпосередньо в рядки.
4. **`привітання("Іван")`:**
    - Це виклик функції привітання з аргументом "Іван".
    - Функція виконується, і на екран виводиться "Привіт, Іван!".
5. **`привітання("Марія")`:**
    - Це ще один виклик функції з іншим аргументом.
    - На екран виводиться "Привіт, Марія!".

Як запустити програму:

1. Збережіть код у файл з розширенням .py, наприклад, привітання.py.
2. Відкрийте командний рядок або термінал.
3. Перейдіть до папки, де ви зберегли файл, за допомогою команди cd.
4. Введіть python привітання.py і натисніть Enter.

Ви побачите результат виконання програми:

Привіт, Іван!
Привіт, Марія!
Цей приклад демонструє, як використовувати функції для організації коду та повторного використання його частин.

Приведемо ще декілька прикладів.

1. **Обчислення площі геометричних фігур:**

   Python

   ```
   def площа_прямокутника(довжина, ширина):
       """Обчислює площу прямокутника."""
       return довжина * ширина

   def площа_кола(радіус):
       """Обчислює площу кола."""
       import math
       return math.pi * радіус**2

   # Використання функцій
   print(f"Площа прямокутника: {площа_прямокутника(5, 10)}")
   print(f"Площа кола: {площа_кола(7)}")
   ```

2. **Перевірка числа на парність:**

   Python

   ```
   def парне_число(число):
       """Перевіряє, чи є число парним."""
       return число % 2 == 0

   # Використання функції
   print(f"Число 10 парне? {парне_число(10)}")
   print(f"Число 7 парне? {парне_число(7)}")
   ```

3. **Обчислення факторіалу числа:**

   Python

   ```
   def факторіал(число):
       """Обчислює факторіал числа."""
       if число == 0:
       return 1
       else:
       return число * факторіал(число - 1)

   # Використання функції
   print(f"Факторіал 5: {факторіал(5)}")
   ```

4. **Перетворення температури з Цельсія в Фаренгейт:**

Python

```
def цельсій_в_фаренгейт(температура):
    """Перетворює температуру з Цельсія в Фаренгейт."""
    return (температура * 9/5) + 32

# Використання функції
print(f"25 градусів Цельсія = {цельсій_в_фаренгейт(25)} градусів Фаренгейта")
```

5. **Пошук найбільшого числа в списку:**

Python

```
def найбільше_число(список):
    """Знаходить найбільше число в списку."""
    максимум = список[0]
    for число in список:
    if число > максимум:
    максимум = число
    return максимум

# Використання функції
список_чисел = [12, 5, 27, 8, 19]
print(f"Найбільше число в списку: {найбільше_число(список_чисел)}")
```

Ці приклади демонструють, як використовувати функції для розв'язання різних задач, від простих обчислень до більш складних операцій.

2.2. Основи Python: Синтаксис, змінні та типи даних

Вивчення будь-якої мови програмування починається з опанування її фундаментальних елементів: синтаксичних правил, механізмів роботи зі змінними та системи типів даних. Опанування цих базових компонентів є необхідною передумовою для розробки складних програмних систем, оскільки вони формують основу для вираження алгоритмічної логіки.

1. Базовий синтаксис: Ключові синтаксичні правила

Синтаксис мови Python характеризується мінімалізмом та орієнтованістю на високу читабельність коду, що сприяє швидкій розробці та спрощує подальшу підтримку програм. Нижче наведено ключові синтаксичні правила, які є основоположними для написання коду на Python.

а) Використання відступів для визначення блоків коду

Однією з фундаментальних та найхарактерніших рис мови програмування Python є використання відступів для структурування коду та визначення логічних блоків. На відміну від мов, що використовують для цих цілей фігурні дужки ({}) або спеціальні

ключові слова, у Python рівень вкладеності інструкцій визначається виключно їхнім горизонтальним вирівнюванням.

Код, що логічно належить до певного керуючого оператора (наприклад, if, for, while), має бути зсунутий вправо на однакову кількість пробілів відносно цього оператора. Офіційним керівництвом зі стилю коду PEP 8 рекомендовано використовувати 4 пробіли на один рівень вкладеності.

Даний підхід не є стилістичною рекомендацією, а становить фундаментальну частину синтаксису мови. Це забезпечує уніфіковане форматування коду, підвищує його читабельність та усуває неоднозначність у визначенні меж логічних блоків.

```
# Приклад коректної структури з відступами
age = 20
if age > 18:
  print("Особа є повнолітньою.")
  if age >= 21:
    # Цей блок є вкладеним відносно попереднього
    print("Особа має право на придбання алкогольних напоїв.")

# Приклад, що призведе до помилки IndentationError
if age > 18:
print("Помилка: відсутній обов'язковий відступ.")
```

б) Використання коментарів для документування коду

Коментарі являють собою текстові анотації в програмному коді, які ігноруються інтерпретатором під час виконання і призначені виключно для розробників.

- Способи реалізації коментарів:
  - Однорядкові коментарі: Починаються із символу # і тривають до кінця рядка.
  - Багаторядкові коментарі (рядки документації): Формально є багаторядковими рядками, що обмежуються потрійними лапками (`"""..."""` або `'''...'''`). При розміщенні на початку модуля, класу або функції вони слугують рядками документації (docstrings), які можуть бути оброблені автоматизованими засобами для генерації технічної документації.
- Призначення коментарів:
  1. Пояснення логіки: Анотація складних або неочевидних фрагментів коду.
  2. Планування розробки: Використання тегів, як-от TODO, для позначення ділянок коду, що потребують доопрацювання.
  3. Тимчасова деактивація коду: Закоментовані рядки не виконуються, що дозволяє тимчасово виключати частини програми для тестування.

в) Чутливість до регістру символів (

Python є мовою, чутливою до регістру. Це означає, що ідентифікатори my_variable, My_Variable та MY_VARIABLE будуть інтерпретовані як три різні, непов'язані змінні. Це правило поширюється на всі іменовані сутності в мові, включаючи імена функцій, класів та модулів.

2. Змінні як іменовані посилання на об'єкти в пам'яті

У контексті Python змінна є не контейнером для даних, а іменованим посиланням, що вказує на об'єкт, який зберігається в пам'яті.

Змінна створюється (ініціалізується) в момент першого присвоєння їй значення за допомогою оператора =. У цей момент створюється об'єкт у пам'яті, а змінна починає на нього посилатися.

```
# Створення змінної 'age', що посилається на об'єкт типу int зі значенням 25
age = 25

# Створення змінної 'user_name', що посилається на об'єкт типу str зі значенням
"Анна"
user_name = "Анна"
```

Принцип динамічної типізації

Однією з ключових характеристик Python, що надає мові значної гнучкості, є динамічна типізація. Суть принципу така - тип змінної не оголошується заздалегідь (на відміну від статично типізованих мов, як C++ чи Java), а визначається автоматично інтерпретатором у момент присвоєння значення. Це дозволяє одному й тому ж посиланню (змінній) послідовно вказувати на об'єкти різних типів.

```
my_data = 100       # my_data посилається на об'єкт типу int
print(type(my_data)) # Виведе: <class 'int'>

my_data = "сто"     # my_data тепер посилається на об'єкт типу str
print(type(my_data)) # Виведе: <class 'str'>
```

Динамічна типізація значно прискорює процес прототипування та проведення дослідницьких експериментів, що є типовим для сфери ШІ. Водночас для великих програмних систем це може підвищувати ризик виникнення помилок під час виконання. Для нівелювання цього недоліку в сучасних версіях Python було впроваджено механізм анотацій типів (type hints), який дозволяє розробнику опціонально вказувати очікувані типи, що покращує читабельність та дозволяє інструментам статичного аналізу виявляти потенційні помилки.

3. Практичне застосування типів даних

Розглянемо практичне застосування раніше описаних теоретичних типів даних у синтаксисі Python.

```
int (цілі числа):
epoch_count = 50
float (числа з плаваючою комою):
learning_rate = 0.001
bool (логічний тип):
training_complete = False
str (рядки):
model_name = "MyFirstClassifier"
```

Перевірка та приведення типів

При роботі з даними, що надходять із зовнішніх джерел (напр., введення користувача, файли), фундаментальною операцією є перетворення типів.

- Функція type(): Дозволяє визначити тип об'єкта, на який посилається змінна.
- Функції приведення типів: int(), float(), str() виконують спробу перетворити передане значення на відповідний тип.

```
user_input = input("Введіть ваш вік: ") # Функція input() завжди повертає об'єкт типу str

# Спроба виконати математичну операцію з рядком призведе до помилки TypeError

# Перетворюємо рядок на ціле число для математичних операцій
try:
    user_age = int(user_input)
    print(f"Через 5 років вам буде {user_age + 5} років.")
except ValueError:
    print("Помилка: введено некоректні дані. Неможливо виконати перетворення типу.")
```

Некоректне поводження з типами даних може призвести до помилок під час виконання, таких як TypeError (операція не підтримується для даного типу) або ValueError (значення не може бути перетворено на цільовий тип). Розуміння та контроль типів даних є запорукою написання надійного та коректного програмного коду.

Отже, синтаксис, механізми роботи зі змінними та система типів даних є трьома фундаментальними компонентами мови Python. Опанування зазначених базових концепцій дозволяє перейти до розробки складних програмних рішень, що здатні вирішувати реальні завдання, зокрема й у сфері штучного інтелекту. Дані принципи є не просто синтаксичними правилами, а інструментами для точного вираження алгоритмічної логіки.

2.3. Базові алгоритмічні структури в Python

Будь-який алгоритм, незалежно від його складності, можна побудувати з трьох базових структур: **слідування**, **розгалуження** та **циклу**. Мова Python надає прості та інтуїтивно зрозумілі інструменти для реалізації кожної з них, що робить її чудовим вибором для вивчення програмування.

1. Слідування (Послідовне виконання)

Слідування — це найпростіша структура, де інструкції (команди) програми виконуються послідовно, одна за одною, у тому порядку, в якому вони написані в коді. Інтерпретатор Python читає ваш файл зверху вниз і виконує кожен рядок, не перестрибуючи і не повертаючись назад. Це природний потік виконання програми.

У Python будь-який код, написаний без умовних операторів чи циклів, є прикладом слідування.

Приклад: Програма, яка запитує ім'я користувача, вітається з ним, а потім повідомляє кількість символів у його імені.

```
# 1. Запитати ім'я користувача і зберегти його у змінну 'name'
print("Програма для аналізу імені.")
name = input("Будь ласка, введіть ваше ім'я: ")

# 2. Створити вітальне повідомлення
greeting = f"Привіт, {name}!"

# 3. Вивести вітання на екран
print(greeting)

# 4. Обчислити довжину імені
name_length = len(name)

# 5. Вивести повідомлення про довжину імені
print(f"Ваше ім'я складається з {name_length} літер.")
print("Програму завершено.")
```

У цьому прикладі кожна команда виконується суворо після завершення попередньої. Порядок є фіксованим, і його зміна призведе до зміни логіки програми.

2. Розгалуження (Умовне виконання)

Розгалуження дозволяє програмі приймати рішення і виконувати різні блоки коду залежно від того, чи є певна умова істинною (True) чи хибною (False). В основі розгалуження в Python лежать оператори if, elif та else. Це ключовий інструмент для створення гнучкої та інтерактивної логіки.

а) Конструкція if

Виконує блок коду, тільки якщо умова є істинною. Варто зазначити, що в Python не тільки True вважається істиною. Будь-яке число, крім 0, та будь-який непустий рядок чи список також інтерпретуються як True.

```
age = 20
has_passport = True

# Перевіряємо, чи вік більше або дорівнює 18
if age >= 18 and has_passport:
   print("Ви повнолітній і маєте паспорт, можете подорожувати за кордон.")

print("Перевірку документів завершено.")
```

б) Конструкція if...else

Надає два шляхи виконання: один, якщо умова істинна, та інший — якщо хибна. Це гарантує, що один з двох блоків коду буде виконано.

```
temperature = 15

if temperature > 20:
   print("Надворі тепло, можна йти в футболці.")
else:
   print("Сьогодні прохолодно, краще одягнути куртку.")
```

в) Конструкція if...elif...else

Використовується, коли є більше двох можливих варіантів. Програма перевіряє умови послідовно і виконує блок коду, що відповідає першій істинній умові. Блок else виконується, якщо жодна з попередніх умов не виявилася істинною.

```
score = 85

if score >= 90:
   grade = "Відмінно (A)"
elif score >= 82:
   grade = "Дуже добре (B)"
elif score >= 75:
   grade = "Добре (C)"
elif score >= 65:
   grade = "Задовільно (D)"
else:
   grade = "Незадовільно (F)"

print(f"Ваша оцінка: {grade}")
```

3. Цикл (Повторне виконання)

Цикл дозволяє виконувати один і той самий блок коду багаторазово. Це надзвичайно корисно для обробки колекцій даних (наприклад, всіх файлів у папці) або повторення дій до досягнення певної мети. У Python є два основних типи циклів: for та while.

а) Цикл for

Цикл for ідеально підходить для ітерації (перебору) елементів у будь-якій ітерабельній послідовності: списку, рядку, кортежі, словнику або діапазоні чисел.

Приклад з перебором списку:

```
fruits = ["яблуко", "банан", "вишня"]

print("Фрукти в наявності:")
for fruit in fruits:
   print(f"- {fruit.capitalize()}") # Робимо першу літеру великою
```

Приклад з використанням range() для повторення дії певну кількість разів:

```
# Виконати блок коду 5 разів
for i in range(1, 6): # range(1, 6) генерує числа від 1 до 5
  print(f"Це повідомлення номер {i}")
```

б) Цикл while

Цикл while виконує блок коду доти, доки задана умова залишається істинною (True). Він ідеальний для ситуацій, коли кількість ітерацій наперед невідома. Критично важливо, щоб змінна, яка перевіряється в умові, змінювалася всередині тіла циклу, інакше ви отримаєте *нескінченний цикл* — поширену помилку, яка змушує програму "зависнути".

Приклад: Валідація вводу користувача.

```
user_input = ""
while user_input.lower() != "так":
  user_input = input("Ви погоджуєтесь з умовами? (введіть 'так' для продовження):
")

print("Дякуємо! Ви погодились з умовами.")
```

в) Керування циклом: break та continue

- break: Негайно та повністю **завершує** виконання циклу.
- continue: Пропускає залишок коду в поточній ітерації та **переходить до наступної**.

Приклад з break: Пошук першого користувача з іменем "admin".

```
users = ["john", "peter", "admin", "mary"]
for user in users:
  print(f"Перевірка користувача: {user}")
  if user == "admin":
    print("Знайдено адміністратора! Пошук зупинено.")
    break # Виходимо з циклу, бо ціль досягнута
```

Приклад з continue. Обробка тільки позитивних чисел.

```
numbers = [10, -5, 20, 0, -15, 30]
for num in numbers:
  if num <= 0:
    continue # Пропускаємо непозитивні числа і переходимо до наступного
  print(f"Обробляємо позитивне число: {num}")
```

2.4. Списки Python для зберігання наборів даних

У програмуванні часто доводиться працювати з наборами однотипних даних: списком користувачів, переліком товарів, результатами вимірювань тощо. Одним з найпростіших, найгнучкіших та найпоширеніших інструментів для зберігання таких наборів у мові Python є список (list). Він слугує фундаментальною основою для багатьох операцій з даними.

Список — це впорядкована, змінна колекція елементів, що робить його надзвичайно універсальним.

- Впорядкована. Кожен елемент має свій унікальний порядковий номер (індекс), що починається з 0. Це означає, що порядок, у якому ви додаєте елементи, зберігається. Перший елемент завжди матиме індекс 0, другий — 1, і так далі. Це критично важливо для даних, де послідовність має значення (наприклад, часові ряди).
- Змінна. Ви можете легко додавати, видаляти та змінювати елементи після створення списку. Ця динамічність дозволяє адаптувати набір даних під час виконання програми.
- Дозволяє дублікати. Список може без проблем містити однакові значення, що необхідно для відображення реальних даних, де повторення є нормою.
- Гнучкий. Може зберігати елементи різних типів даних (числа, рядки, інші списки тощо) в межах однієї колекції.

Створити список дуже просто:

```
# Порожній список для подальшого наповнення
products = []

# Список з початковими даними
temperatures = [18.5, 20.2, 19.0, 22.1, 17.8]
```

Простий список підходить для зберігання одного ряду даних (наприклад, температур). Але що робити, коли дані мають структуру, подібну до таблиці з рядками та стовпцями?

Найпоширенішими підходами є:

1. Список списків (вкладені списки)
2. Список словників

1. Список списків

Кожен елемент зовнішнього списку — це інший список, який представляє один рядок даних.

Переваги: Компактний запис.
Недоліки: Крихкість та низька читабельність. Код стає залежним від "магічних чисел" (індексів). Якщо в майбутньому порядок даних зміниться (наприклад, ID і ім'я поміняються місцями), код зламається, але помилка може бути неочевидною. Якщо структура даних зміниться (наприклад, додасться новий стовпець), доведеться переглядати та виправляти всі індекси в коді. Приклад: список співробітників, де кожен запис містить ID, ім'я та посаду.

```
# Кожен вкладений список - це один співробітник
# [ID, Ім'я, Посада]
employees = [
   [101, "Олена Петренко", "Маркетолог"],
   [102, "Іван Ковальчук", "Розробник"],
   [103, "Марія Сидоренко", "Дизайнер"]
]

# Доступ до даних
first_employee_name = employees[0][1] # employees[0] -> перший рядок; [1] ->
другий елемент рядка
second_employee_position = employees[1][2] # "Розробник"

print(f"Ім'я першого співробітника: {first_employee_name}"
```

2. Список словників

Це більш сучасний та рекомендований підхід. Зовнішній список містить словники, де кожен словник — це один рядок даних. Ключі словника виступають у ролі назв стовпців.

Переваги: Читабельність і надійність. Код стає самодокументованим, оскільки назви ключів ("name", "position") чітко пояснюють, які дані ми отримуємо. На відміну від доступу за індексом, такий код не зламається, якщо до словника додадуться нові поля або зміниться їхній порядок. Це значно знижує ризик помилок і полегшує підтримку коду в майбутньому.
Недоліки: Трохи більш громіздкий запис у порівнянні зі списком списків.
Приклад: Той самий список співробітників.

```
employees_as_dicts = [
   {"id": 101, "name": "Олена Петренко", "position": "Маркетолог"},
   {"id": 102, "name": "Іван Ковальчук", "position": "Розробник"},
   {"id": 103, "name": "Марія Сидоренко", "position": "Дизайнер"}
]

# Доступ до даних значно читабельніший
first_employee_name = employees_as_dicts[0]["name"]
second_employee_position = employees_as_dicts[1]["position"]

print(f"Ім'я першого співробітника: {first_employee_name}")
```

Обробка даних у списках. Незалежно від структури, для обробки набору даних найчастіше використовується цикл for.

Приклад: Вивести імена всіх розробників зі списку словників.

```
print("\nСписок розробників:")
developer_names = []
for employee in employees_as_dicts:
```

```
    if employee["position"] == "Розробник":
        developer_names.append(employee["name"])

print(developer_names) # Виведе: ['Іван Ковальчук']
```

Для більш компактного запису подібних операцій (фільтрації та трансформації) в Python часто використовують генератори списків (list comprehensions):

```
# Альтернативний, більш "пайтонічний" спосіб
# Структура: [вираз for елемент in список if умова]
developer_names_comp = [employee["name"] for employee in employees_as_dicts if employee["position"] == "Розробник"]
print(developer_names_comp) # Виведе: ['Іван Ковальчук']
```

Списки використовуються для наступних операцій.

- Для зберігання невеликих та середніх за обсягом наборів даних. Для десятків, сотень або навіть тисяч записів списки працюють чудово. Однак для сотень тисяч або мільйонів записів їх ефективність знижується, особливо під час пошуку конкретного елемента, оскільки Python змушений переглядати кожен елемент послідовно (лінійний пошук).
- Коли потрібна проста, вбудована структура без встановлення додаткових бібліотек. Це ідеально для невеликих скриптів, швидких обчислень та завдань, де підключення зовнішніх залежностей є надлишковим.
- На початкових етапах розробки та для прототипування. Списки дозволяють швидко накидати структуру даних і перевірити логіку роботи програми, перш ніж переходити до більш складних інструментів.

Для великих обсягів даних та складних аналітичних задач (фільтрація, групування, агрегація) варто використовувати спеціалізовані бібліотеки, такі як Pandas, яка надає значно більш потужні та оптимізовані інструменти. Однак розуміння того, як працюють списки, є фундаментальною основою для будь-якого розробника на Python.

2.5. Кортежі для незмінних наборів даних

У Python, окрім списків (list), існує ще одна вбудована структура для зберігання колекцій — кортеж (tuple). Головна відмінність кортежу від списку полягає в його незмінності (immutability). Після створення кортежу ви не можете додати, видалити або змінити його елементи. Ця, на перший погляд, обмежувальна властивість насправді є його найбільшою перевагою в певних задачах, оскільки забезпечує передбачуваність та безпеку даних.

Кортеж — це впорядкована, незмінна колекція елементів. Його створюють за допомогою круглих дужок ().

```
# Порожній кортеж
empty_tuple = ()
```

```
# Кортеж з різними типами даних
# Часто використовується для представлення однієї структури, наприклад, точки на графіку
point = (10, 20, "A")

# Створення кортежу без дужок (Python автоматично розпізнає його)
# Це називається "упаковкою кортежу" (tuple packing)
config = "localhost", 8080, True

# Важливо: створення кортежу з одного елемента
# Потрібно поставити кому після елемента, щоб відрізнити його від простого виразу в дужках
single_element_tuple = (42,)
not_a_tuple = (42) # Це буде просто число (int) 42
```

Використання кортежів замість списків має кілька ключових переваг, пов'язаних саме з їхньою незмінністю.

1. Цілісність та безпека даних

Це найважливіша причина. Якщо ви маєте набір даних, який не повинен змінюватися протягом роботи програми (наприклад, конфігураційні налаштування, координати, RGB-коди кольорів), використання кортежу гарантує, що ці дані залишаться недоторканими. Це захищає від цілого класу випадкових помилок, коли функція ненавмисно змінює дані, які передаються їй як аргумент.

Приклад**.** Зберігання RGB кольорів.

```
# Визначаємо основні кольори як кортежі
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

def process_color(color_data):
    # Уявімо, що тут відбувається якась помилка, і ми намагаємося змінити дані
    # color_data[0] = 0 # Якщо color_data - це список, він зміниться!
    pass

# Якщо передати кортеж, будь-яка спроба його змінити всередині функції викличе помилку
# RED[0] = 128  # Це призведе до TypeError: 'tuple' object does not support item assignment
process_color(RED) # Ваші вихідні дані залишаються в безпеці
```

Важливо розуміти, що незмінність кортежу стосується посилань на об'єкти, які він зберігає. Якщо один з цих об'єктів є змінним (наприклад, список), то його вміст можна змінити.

```
# Кортеж містить список
user_roles = (123, 'admin', ['read', 'write'])

# Змінити сам кортеж не можна:
# user_roles[1] = 'guest' # TypeError

# Але можна змінити список всередині кортежу!
user_roles[2].append('delete')

print(user_roles) # Виведе: (123, 'admin', ['read', 'write', 'delete'])
```

Це показує, що кортеж гарантує незмінність своєї структури, але не вмісту змінних елементів.

2. Продуктивність

Оскільки кортежі незмінні, Python може застосовувати до них певні оптимізації. Інтерпретатор знає, що розмір кортежу ніколи не зміниться, тому може виділити пам'ять більш ефективно. Зазвичай вони займають трохи менше пам'яті та створюються швидше, ніж списки. Хоча ця перевага існує, головною причиною для вибору кортежу майже завжди є безпека та цілісність даних, а не мікрооптимізація.

3. Використання як ключів у словниках

Ключем у словнику (dict) може бути тільки **незмінний (хешований)** об'єкт. Це пов'язано з тим, що словник обчислює хеш-значення ключа, щоб швидко знаходити відповідне йому значення. Якби ключ можна було змінити, його хеш також би змінився, і словник "загубив" би дані. Списки не можна використовувати як ключі, а кортежі — можна.

Приклад**.** Зберігання населення для міст, визначених парою (широта, довгота).

```
city_population = {
   (49.44, 32.06): 280000,  # Черкаси
   (50.45, 30.52): 2960000, # Київ
}

# Доступ до даних за допомогою кортежу-ключа
kyiv_population = city_population[(50.45, 30.52)]
print(f"Населення Києва: {kyiv_population}"
```

Хоча кортежі не можна змінювати, ви можете легко отримувати доступ до їхніх елементів та обробляти їх. Як і в списках, **доступ до елементів відбувається за їхнім індексом**, включаючи від'ємні індекси для доступу з кінця.

```
config = ("localhost", 8080, True)
host = config[0]      # "localhost"
port = config[1]      # 8080
is_active = config[-1] # True (останній елемент)
```

Розпакування (Unpacking) - це одна з найпотужніших та найелегантніших можливостей кортежів. Ви можете присвоїти елементи кортежу одразу кільком змінним, що робить код значно чистішим.

```
# Дані про користувача: (id, username, email)
user_data = (101, "admin_user", "admin@example.com")

# Розпаковуємо кортеж у змінні
user_id, username, email = user_data

print(f"ID користувача: {user_id}, Ім'я: {username}")
```

Цей підхід часто використовується для повернення кількох значень із функції. Замість того щоб повертати словник чи список, функція може повернути кортеж, який одразу ж розпаковується у змінні. Існує також розширене розпакування за допомогою оператора *, який збирає всі "зайві" елементи у список:

```
# Отримання першого, останнього та всіх інших елементів
record = (1, 2, 3, 4, 5, 6)
first, *middle, last = record

print(f"Перший: {first}")   # 1
print(f"Середні: {middle}") # [2, 3, 4, 5]
print(f"Останній: {last}") # 6
```

```
def get_user_info(user_id):
# Уявімо, що тут йде запит до бази даних
name = "Олена"
status = "активний"
return name, status # Повертається кортеж ('Олена', 'активний')
user_name, user_status = get_user_info(1)
```

Коли обирати кортеж, а коли — список? Вибір між цими двома структурами часто є семантичним — він показує ваш намір як розробника. Використовуйте `tuple` (кортеж), коли:
Ви працюєте з набором, де кожен елемент має своє конкретне місце та значення (наприклад, `(ім'я, прізвище, вік)`).

- Набір даних є сталим і не повинен змінюватися (координати, налаштування, історичні дані).
  Вам потрібен ключ для словника, що складається з кількох частин.
  Ви повертаєте з функції кілька значень.

- Вам критично важлива цілісність даних і захист від випадкових змін.

Використовуйте `list` (список), коли:

Ви працюєте з набором однотипних елементів (наприклад, список імен, список температур).

Вам потрібно додавати, видаляти або змінювати елементи "на місці". Ви працюєте з колекцією, розмір якої динамічно змінюється (наприклад, список завдань для виконання).

Порядок елементів може змінюватися (наприклад, в результаті сортування).

Отже, кортежі — це не просто "списки, які не можна змінити". Розглядайте список як контейнер для набору схожих речей, а кортеж — як єдиний, цілісний запис із фіксованою структурою. Такий підхід робить ваш код більш надійним, безпечним та зрозумілим для інших розробників.

2.6. Словники та множини: Ключові інструменти в ШІ

У Python існують дві надзвичайно потужні та універсальні структури даних, що базуються на хеш-таблицях: словники (`dict`) та множини (`set`). Їхня архітектура робить їх ідеальним інструментом не лише для простого зберігання даних, але й для вирішення складних завдань у таких сферах, як веб-розробка, аналіз даних та обробка природної мови (NLP).

1. Словники як основа для роботи з даними у форматі JSON

Сучасний ШІ живе у світі, де дані постійно передаються між різними сервісами. Модель може отримувати дані від веб-додатка, звертатися до зовнішнього API за додатковою інформацією або зберігати свої результати у форматі, зрозумілому для інших систем. Стандартом де-факто для такого обміну є JSON (JavaScript Object Notation).

Структура об'єкта JSON є практично ідентичною до структури словника Python, що робить їх ідеальною парою.

- Об'єкт JSON (`{...}`) відповідає словнику Python (`dict`).
- Масив JSON (`[...]`) відповідає списку Python (`list`).

Python має вбудовану бібліотеку `json` для легкої конвертації між цими двома форматами.

Щоб зрозуміти, чому цей зв'язок такий важливий, давайте розшифруємо два ключові терміни:

- Парсер (Parser): Уявіть, що ви отримали довгий текстовий рядок: `{"name": "Іван", "age": 30}`. Для вас це просто текст. Парсер — це спеціальна програма або компонент, який "читає" цей текст, перевіряє, чи він написаний за правилами формату (в нашому випадку JSON), і розбирає його на значущі частини. У цьому прикладі `json.loads()` виступає в ролі парсера. Він аналізує рядок, знаходить ключі "name" та "age", їхні значення, і перетворює все це на структурований об'єкт, з яким може працювати Python.
- Нативний (Native): Це слово в програмуванні означає "рідний" або "вбудований" для певної мови. Словник (`dict`) — це нативний тип даних у Python. Вам не потрібно встановлювати додаткові бібліотеки, щоб його використовувати, він є частиною самої мови. Той факт, що парсер JSON перетворює дані одразу в нативну для Python структуру (словник), робить роботу неймовірно зручною.

а) Перетворення словника Python у JSON (Серіалізація)

Для перетворення словника у рядок JSON використовується функція `json.dumps()` (dump string). Цей процес, коли об'єкт з пам'яті перетворюється у формат для передачі чи зберігання, називається серіалізацією.

```
import json

# Дані про користувача у вигляді словника Python
user_data = {
    "id": 101,
    "name": "Марія Іваненко",
    "is_active": True,
    "courses": ["Python Basics", "Web Development"],
    "profile": {
        "city": "Київ",
        "experience_years": 3
    },
    "last_login": None # Python's None перетворюється на null в JSON
}

# Серіалізуємо словник у рядок JSON
json_string = json.dumps(user_data, ensure_ascii=False, indent=4)

print("--- Рядок у форматі JSON ---")
print(json_string)
```

б) Перетворення JSON у словник Python (Десеріалізація)

Для зворотного процесу — парсингу рядка JSON і перетворення його у словник — використовується функція `json.loads()` (load string). Цей процес називається десеріалізацією.

```
# Уявімо, що цей рядок ми отримали ззовні
incoming_json = """
{
    "id": 102,
    "name": "Петро Коваль",
    "is_active": false,
    "courses": ["Data Science", "Machine Learning"]
}
"""
try:
    # Парсер json.loads() перетворює рядок на нативний словник Python
    parsed_data = json.loads(incoming_json)
    print("\n--- Словник Python після парсингу JSON ---")
    print(parsed_data)

    # Тепер ми можемо легко працювати з даними
    print(f"\nІм'я користувача: {parsed_data['name']}")
    print(f"Перший курс: {parsed_data.get('courses')[0]}")

except json.JSONDecodeError as e:
```

```
    print(f'Помилка декодування JSON: {e}')
```

2. Словники як лічильники слів у задачах NLP

Підрахунок частоти слів — це фундаментальне завдання в обробці природної мови (NLP). Він використовується для аналізу текстів, визначення ключових слів та як перший крок для більш складних моделей. Словники ідеально підходять для цієї задачі.
Приклад.

```
import re

text = "Сонце світить, і річка тече. Річка блищить на сонці, і це чудово."
# 1. Нормалізація тексту
cleaned_text = re.sub(r'[^\w\s]', '', text).lower()
# 2. Розбиття на слова (токенізація)
words = cleaned_text.split()

# 3. Підрахунок за допомогою словника
word_counts = {}
for word in words:
    word_counts[word] = word_counts.get(word, 0) + 1

print("\n--- Частота слів у тексті ---")
print(word_counts)
```

Результат. `{'сонце': 2, 'світить': 1, 'і': 2, 'річка': 2, ...}`

3. Множини: інструмент для роботи з унікальністю

Множина (`set`) — це невпорядкована колекція унікальних елементів. Як і словники, вона базується на хеш-таблицях, що забезпечує надзвичайно швидку перевірку наявності елемента (O(1)).

Застосування множин в ШІ наступне.

- Створення вокабулярів та дедуплікація: Це найпоширеніше застосування. Якщо вам потрібно отримати список усіх унікальних слів з тексту або унікальних ID користувачів зі списку, найпростіший спосіб — перетворити список на множину.
  words = ["мова", "python", "мова", "шІ", "python"]

```
unique_words = set(words)
print(unique_words) # Виведе: {'python', 'шІ', 'мова'}
```

- Порівняння наборів даних. Множини підтримують потужні математичні операції. Уявіть, що у вас є два набори ознак, які використовують дві різні моделі. Ви можете легко знайти:
  - Спільні ознаки (перетин): `features1 & features2`
  - Усі унікальні ознаки (об'єднання): `features1 | features2`
  - Ознаки, що є в першому, але не в другому наборі (різниця): `features1 - features2`
- Швидка фільтрація. Перевірка `if item in my_set:` працює значно швидше, ніж `if item in my_list:`, якщо набір елементів для перевірки великий. Це корисно, наприклад, для фільтрації стоп-слів у задачах NLP.

Чому опанування цих структур є критично важливим? Окрім перелічених вище прикладів, словники та множини є незамінними для:

- Керування конфігураціями та гіперпараметрами: Будь-який ШІ-проєкт має безліч налаштувань. Зберігання їх у словнику дозволяє централізовано керувати експериментами.
- Ефективного представлення розріджених даних: У задачах NLP вектор, що представляє речення, може мати десятки тисяч елементів, але лише кілька з них будуть ненульовими. Зберігати такий вектор у вигляді словника `{індекс_слова: кількість}` значно ефективніше з точки зору пам'яті.
- Швидкого пошуку та перевірки унікальності: Завдяки реалізації на хеш-таблицях, ці структури забезпечують продуктивність, яка є критично важливою при роботі з великими обсягами даних.

Таким чином, словники та множини виступають у ролі універсальних "перекладачів" та організаторів даних. Опанування роботи з ними є абсолютно необхідним навиком для будь-якого розробника на Python, який працює у сфері штучного інтелекту.

2.7. Множини: основа для роботи з унікальними елементами

У Python, крім списків та кортежів, існує ще одна потужна структура даних для роботи з колекціями — множина (set). Її головна ідея — це компроміс: ви відмовляєтеся від порядку та індексів на користь двох ключових переваг: гарантованої унікальності елементів та надзвичайної швидкості операцій.

Множина — це невпорядкована, змінна колекція унікальних, хешованих елементів.

- Унікальність: Кожен елемент у множині може існувати лише в одному екземплярі. При спробі додати елемент, який вже є в множині, нічого не відбудеться, і помилки не виникне.
- Невпорядкованість: Елементи не мають сталого порядку чи індексу. Це пов'язано з тим, що множини реалізовані через хеш-таблиці, де положення елемента визначається його хешем, а не порядком додавання. Тому при виводі на екран їхній порядок може бути довільним і змінюватися між різними запусками програми.
- Змінність: Ви можете додавати (add()) та видаляти елементи з множини після її створення. Для видалення існують два методи: remove() видалить елемент, але викличе помилку KeyError, якщо елемента не існує; discard() також видаляє елемент, але не викличе помилку, якщо його немає.
- Хешованість елементів: Елементами множини можуть бути лише незмінні (immutable) типи даних, такі як числа, рядки, кортежі. Списки, словники чи інші множини не можуть бути елементами множини, оскільки їх вміст можна змінити, що унеможливлює стабільне хешування.

Послідовність - абстрактна структура, у якої множина елементів є такою, що для кожного елемента, крім першого і останнього, є попередній і наступний елементи.

Створення множини:

```
# Створення множини за допомогою фігурних дужок
unique_numbers = {1, 2, 3, 4, 5}

# Створення з ітерабельного об'єкта (наприклад, списку) за допомогою set()
# Дублікати будуть автоматично видалені
my_list = [1, 'a', 2, 'b', 1, 'a']
unique_elements = set(my_list)
print(f"Множина з унікальних елементів списку: {unique_elements}") # Виведе: {1, 2, 'a', 'b'}

# Важливо: створення порожньої множини
empty_set = set()
# empty_braces = {} # Це створить порожній словник, а не множину!
```

Одним із найпоширеніших завдань в обробці природної мови (NLP) є створення вокабуляра (словника) — повного переліку всіх унікальних слів, що зустрічаються в тексті або наборі текстів. Множини ідеально підходять для цієї задачі. Створений вокабуляр є основою для подальших операцій, таких як векторизація тексту (наприклад, модель "мішок слів").

Алгоритм створення вокабуляра:

1. Нормалізація тексту: Приведення всіх слів до нижнього регістру та видалення знаків пунктуації. Це гарантує, що "Слово" та "слово." будуть вважатися однаковими.
2. Токенізація: Розбиття тексту на окремі слова (токени).
3. Створення множини: Передача списку токенів у конструктор set(), який автоматично залишить лише унікальні слова.

Приклад:

```
text_data = """
Python є чудовою мовою для аналізу даних.
Використовуйте Python для швидкого старту.
Мова Python проста та гнучка.
"""

# 1. Нормалізація
normalized_text = text_data.lower().replace('.', '').replace('\n', ' ')

# 2. Токенізація
words = normalized_text.split()
print(f"Список всіх слів (з повтореннями): \n{words}\n")

# 3. Створення вокабуляра за допомогою множини
vocabulary = set(words)
print(f"Вокабуляр (унікальні слова): \n{vocabulary}")
```

```
# Розмір вокабуляра - це кількість унікальних слів
print(f"\nКількість унікальних слів: {len(vocabulary)}")
```

Завдяки множині ми отримали повний перелік унікальних слів лише одним рядком коду (set(words)), що є надзвичайно ефективним.

Основні операції та переваги множин

1. Швидка перевірка наявності

Множини реалізовані на основі хеш-таблиць, тому перевірка, чи містить множина певний елемент, виконується в середньому за константний час O(1). Це схоже на пошук слова у словнику за самим словом (дуже швидко), на відміну від пошуку у списку, який схожий на читання книги з першої сторінки в пошуках потрібного слова (час O(n), залежить від розміру).

```
# Уявімо, що в нас мільйон унікальних слів у вокабулярі
large_vocabulary = {'python', 'мова', 'дані', ...} # та багато інших

# Ця перевірка буде майже миттєвою, незалежно від розміру множини
if 'python' in large_vocabulary:
    print("Слово 'python' є у вокабулярі.")
```

Це робить множини ідеальним вибором для фільтрації або валідації даних на основі великого набору правил чи значень. Подібна перевірка у списку з мільйона елементів вимагала б перегляду всіх елементів у найгіршому випадку, що було б у тисячі разів повільніше.

2. Математичні операції над множинами

Множини підтримують потужні математичні операції, які дозволяють легко порівнювати та комбінувати колекції.

- Об'єднання (| або .union()). Створює нову множину з усіма елементами з обох множин.
- Перетин (& або .intersection()). Створює множину лише зі спільних елементів.
- Різниця (- або .difference()). Створює множину з елементів, які є в першій, але відсутні в другій.
- Симетрична різниця (^ або .symmetric_difference()). Створює множину з елементів, які є в одній з множин, але не в обох одночасно.

Приклад. Аналіз ключових слів у двох текстах.

```
text1_words = {'python', 'дані', 'аналіз', 'машинне-навчання'}
text2_words = {'python', 'веб-розробка', 'фреймворки', 'аналіз'}

# Усі унікальні ключові слова з обох текстів
all_keywords = text1_words | text2_words
print(f"Усі ключові слова: {all_keywords}")

# Спільні ключові слова
```

```
common_keywords = text1_words & text2_words
print(f"Спільні ключові слова: {common_keywords}") # {'python', 'аналіз'}

# Ключові слова, унікальні для кожного тексту
unique_keywords = text1_words ^ text2_words
print(f"Унікальні для кожного тексту слова: {unique_keywords}")
```

Отже, множини (set) є незамінним інструментом у Python, коли потрібно працювати з унікальними даними. Вони забезпечують:

- Автоматичне видалення дублікатів.
- Надзвичайно швидку перевірку наявності елементів.
- Потужні операції для порівняння та комбінування наборів даних.

Завдяки цим властивостям, множини є ідеальним вибором для таких завдань, як дедуплікація даних, створення вокабулярів у NLP, фільтрація елементів та реалізація складної логіки на основі приналежності до груп.

2.8. Дерева та їх застосування в ШІ

Дерева є однією з найважливіших нелінійних структур даних у комп'ютерних науках. Вони ідеально підходять для представлення ієрархічних відношень, що робить їх незамінними для вирішення широкого кола завдань, від організації файлових систем до побудови складних моделей штучного інтелекту.

Дерево — це ієрархічна структура, що складається з вузлів, з'єднаних ребрами, і не містить циклів. Бінарне дерево (Binary Tree) — це специфічний тип дерева, де кожен вузол може мати не більше двох нащадків: лівого та правого.

- Корінь (Root): Найвищий вузол у дереві.
- Батько (Parent): Вузол, що має нащадків.
- Нащадок (Child): Вузол, що має батька.
- Лист (Leaf): Вузол, що не має нащадків.

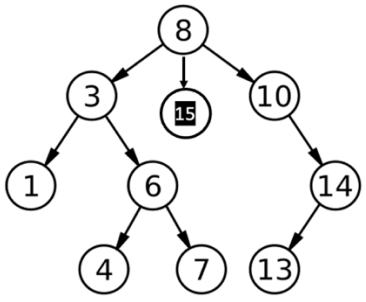


Рис.12. Приклад дерева

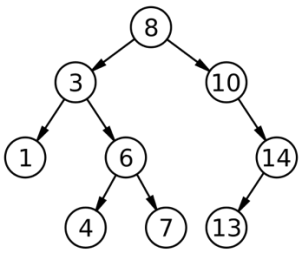


Рис.13. Приклад бінарного дерева

Найпоширеніший спосіб представити дерево в пам'яті — це використання вузлів (nodes), які є об'єктами або структурами. Кожен вузол містить:

1. Дані (Data): Значення, що зберігається у вузлі.
2. Посилання (Pointers/References): Вказівники на його нащадків. У бінарному дереві це зазвичай left та right.

```
# Класичний приклад реалізації вузла дерева в Python
class TreeNode:
    def __init__(self, value):
        self.value = value      # Дані вузла
        self.left = None        # Посилання на лівого нащадка
        self.right = None       # Посилання на правого нащадка
```

Така структура дозволяє динамічно будувати дерево, зв'язуючи об'єкти-вузли між собою.

Щоб відвідати кожен вузол дерева, використовуються спеціальні алгоритми обходу. Існує два основні підходи.

а) Обхід у глибину (Depth-First Search - DFS)

Ця стратегія полягає у дослідженні однієї гілки настільки глибоко, наскільки це можливо, перш ніж повертатися. Зазвичай реалізується за допомогою рекурсії.

- Прямий порядок (Pre-order): Батько -> Лівий -> Правий. Спочатку обробляється поточний вузол, потім його ліве піддерево, потім праве.
- Центрований порядок (In-order): Лівий -> Батько -> Правий. Обробляється ліве піддерево, потім поточний вузол, потім праве. *Для бінарних дерев пошуку цей обхід повертає елементи у відсортованому порядку.*
- Зворотний порядок (Post-order): Лівий -> Правий -> Батько. Спочатку обробляються обидва піддерева, і лише потім — сам вузол.

б) Обхід у ширину (Breadth-First Search - BFS)

Ця стратегія досліджує дерево рівень за рівнем. Спочатку відвідується корінь, потім усі його прямі нащадки, потім їхні нащадки і так далі. Для реалізації використовується черга (Queue).

Будь-яке дерево, де вузол може мати довільну кількість нащадків, можна ефективно представити у вигляді бінарного дерева за схемою "лівий нащадок — правий брат".

- Посилання left у бінарному дереві вказує на першого нащадка оригінального вузла.
- Посилання right вказує на наступного брата (сусіднього вузла на тому ж рівні ієрархії). Це дозволяє уніфікувати обробку дерев різної структури.

Дерева застосовуються в алгоритмах пошуку. Бінарне дерево пошуку (BST) - це спеціальний вид бінарного дерева, де для кожного вузла виконується умова: усі значення в його лівому піддереві є меншими за його власне значення, а всі значення в правому — більшими.

Ця властивість дозволяє реалізувати дуже швидкий пошук. На кожному кроці, порівнюючи шукане значення з поточним вузлом, ми можемо відкинути половину дерева, що залишилася, досягаючи часової складності O(log n) для збалансованого дерева.

Операції включення і виключення з бінарного дерева пошуку

- Включення (Insertion): Новий елемент завжди вставляється як лист. Ми починаємо пошук з кореня, і рухаємося вліво або вправо, доки не знайдемо порожнє місце, що задовольняє умові BST.
- Виключення (Deletion): Це складніша операція, яка розглядає три випадки:
    1. Вузол є листом: Просто видаляється.
    2. Вузол має одного нащадка: Вузол замінюється своїм нащадком.
    3. Вузол має двох нащадків: Вузол замінюється або його in-order наступником (найменшим елементом у правому піддереві), або in-order попередником (найбільшим елементом у лівому піддереві). Потім цей наступник/попередник видаляється зі свого початкового місця.

Види бінарних дерев та проблема збалансованості

- Повне (Full): Кожен вузол має або 0, або 2 нащадки.
- Завершене (Complete): Всі рівні, крім, можливо, останнього, повністю заповнені, а останній заповнюється зліва направо (ця властивість важлива для пірамід).
- Ідеальне (Perfect): Повне дерево, де всі листя знаходяться на одному рівні.
- Вироджене (Degenerate): Кожен вузол має лише одного нащадка. Таке дерево перетворюється на зв'язний список, і ефективність операцій падає до O(n).

Щоб уникнути виродження, використовуються дерева, що самостійно балансуються.

Червоно-чорне дерево (Red-Black Tree)

Це один з найпоширеніших типів збалансованого бінарного дерева пошуку. Кожен вузол має додатковий атрибут — колір (червоний або чорний). Після кожної операції вставки або видалення дерево перевіряє дотримання набору правил червоно-чорного дерева (наприклад, "корінь завжди чорний", "у червоного вузла нащадки — чорні", "будь-який шлях від кореня до листя містить однакову кількість чорних вузлів"). Якщо правила порушені, дерево виконує повороти та перефарбовування вузлів, щоб відновити баланс.

Хоча воно не є ідеально збалансованим, як АВЛ-дерево, воно гарантує, що найдовший шлях не буде більш ніж удвічі довшим за найкоротший, що забезпечує складність O(log n) і вимагає менше операцій для підтримки балансу. Саме тому воно часто використовується в реалізації словників та інших асоціативних масивів у стандартних бібліотеках мов програмування.

У галузі штучного інтелекту, метою якої є розробка систем, здатних до аналізу та прийняття рішень, фундаментальні структури даних відіграють визначальну роль. Серед них дерева займають особливе місце, оскільки вони дозволяють моделювати дані не як невпорядковані набори значень, а як комплексні системи, що визначаються ієрархічними та мережевими відношеннями. Такий підхід є основою для багатьох сучасних досягнень у сфері ШІ.

Ключове застосування деревних структур у класичному штучному інтелекті пов'язане з деревами рішень (Decision Trees). Дерево рішень є моделлю машинного навчання, що відрізняється високим рівнем інтерпретованості та імітує процес прийняття рішень людиною шляхом декомпозиції складної проблеми на послідовність простих бінарних питань.

- Структурні елементи:
  - Кореневий вузол: Представляє початкову ознаку, що забезпечує найбільш ефективний розподіл даних.
  - Гілки: Відображають результати перевірки умов або відповіді на питання.
  - Внутрішні вузли: Містять додаткові ознаки для подальшого уточнення класифікації.
  - Листя (термінальні вузли): Містять кінцевий прогноз або класифікаційну мітку.

Фундаментальне значення для ШІ:

1. Інтерпретованість: На відміну від складних нейронних мереж, що часто функціонують як непрозорі моделі («чорні скриньки»), внутрішня логіка дерева рішень є прозорою та доступною для аналізу. Ця властивість є критично важливою у сферах, що вимагають обґрунтування результатів, зокрема в медицині, фінансах та юриспруденції.
2. Обчислювальна ефективність: Алгоритми побудови дерев рішень є відносно простими в реалізації та не потребують значних обчислювальних ресурсів для навчання, що робить їх ефективним інструментом для широкого кола завдань класифікації та регресії.
3. Основа для ансамблевих методів: Хоча поодинокі дерева схильні до перенавчання, вони слугують базовими елементами для значно більш потужних та стійких моделей:
   - Випадковий ліс (Random Forest): Даний метод використовує ансамбль дерев, кожне з яких побудоване на випадковій підвибірці даних. Кінцевий прогноз формується шляхом агрегації результатів (наприклад, мажоритарного голосування), що підвищує точність і зменшує дисперсію моделі.
   - Градієнтний бустинг (Gradient Boosting): Цей підхід передбачає послідовну побудову дерев, де кожна наступна модель навчається мінімізувати залишкову помилку попередніх. Це дозволяє створювати високоточні прогнозні системи.

Окрім дерев рішень, деревні структури застосовуються в алгоритмах пошуку, наприклад, в ігровому ШІ (алгоритм minimax), де вузли представляють стани гри, а ребра — можливі ходи.

2.9. Графи в ШІ. Моделювання складних взаємозв'язків

Якщо дерева моделюють строгі ієрархії, то графи є значно більш гнучкою та універсальною структурою даних, призначеною для представлення складних мереж та довільних зв'язків. Від соціальних мереж до логістичних маршрутів — графи дозволяють нам моделювати та аналізувати величезну кількість систем реального світу.

Граф — це абстрактна математична структура, що складається з двох основних елементів:

- Вершини (або вузли, nodes, vertices): Представляють окремі об'єкти.
- Ребра (або зв'язки, edges): Представляють зв'язки між парами вершин.

Графи можуть бути:

- Неспрямованими (Undirected): Якщо ребро між вершинами A і B означає, що зв'язок є двостороннім (як дружба у Facebook).
- Спрямованими (Directed): Якщо ребра мають напрямок від однієї вершини до іншої (як підписка в Instagram). Такі графи також називають орграфами.
- Незваженими (Unweighted): Усі ребра вважаються рівноцінними.
- Зваженими (Weighted): Кожне ребро має певну "вагу" або "вартість" (наприклад, відстань між містами на карті).

Існує два основні способи представити граф у програмі:

- Матриця суміжності (Adjacency Matrix) - це двовимірний масив розміром V x V (де V — кількість вершин). Елемент matrix[i][j] дорівнює 1 (або вазі ребра), якщо між вершинами i та j є ребро, і 0 — якщо немає.
    - Переваги - дуже швидка перевірка наявності ребра між двома вершинами (O(1)).
    - Недоліки - вимагає багато пам'яті ($O(V^2)$), що є неефективним для розріджених графів (графів, де кількість ребер значно менша за максимально можливу).
- Список суміжності (Adjacency List) - це масив або словник, де для кожної вершини зберігається список її безпосередніх сусідів.
    - Переваги - ефективне використання пам'яті (O(V + E), де E — кількість ребер), ідеально підходить для розріджених графів.
    - Недоліки - неревірка наявності ребра є трохи повільнішою (O(k), де k — кількість сусідів вершини).

```
# Приклад подання графа списком суміжності в Python
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

Щоб відвідати всі вершини графа, використовуються дві основні стратегії. Ключовою відмінністю від обходу дерев є необхідність відстежувати відвідані вершини (зазвичай за допомогою множини set), щоб уникнути нескінченних циклів.

Алгоритм пошуку у ширину (BFS - Breadth-First Search) досліджує граф "рівень за рівнем". Він починає з початкової вершини, потім відвідує всіх її сусідів, потім сусідів цих сусідів, і так далі. Для своєї роботи він використовує чергу (Queue). BFS завжди знаходить найкоротший шлях за кількістю ребер у незваженому графі.

Алгоритм пошуку у глибину (DFS - Depth-First Search) намагається пройти "вглиб" графа настільки, наскільки це можливо, рухаючись по одній гілці. Дійшовши до кінця гілки, він повертається назад і досліджує іншу. Для своєї роботи він використовує **стек (Stack)**, що найчастіше реалізується за допомогою рекурсії.

Інші типові задачі на графах

Топологічне сортування

Топологічне сортування (Topological Sort) — це спеціальний алгоритм, який може працювати тільки з системами, де є чіткі залежності без циклів (тобто з DAG). Назва “Спрямований Ациклічний Граф” (DAG) звучить складно, але її легко зрозуміти, якщо розбити на частини:

- Граф: Уявіть собі набір точок (вершин) та ліній (ребер), що їх з'єднують. Це і є граф.
- Спрямований (Directed): Це означає, що зв'язки між точками мають напрямок, як вулиці з одностороннім рухом. Ви можете рухатися від точки А до точки Б, але не навпаки (якщо немає іншого ребра Б -> А).
- Ациклічний (Acyclic): Це найважливіша частина. Вона означає, що у графі немає циклів. Тобто, якщо ви почнете рух з будь-якої точки і будете слідувати за стрілками, ви ніколи не зможете повернутися у початкову точку. Немає "кругового руху" чи "зациклень".

Уявіть собі список завдань для приготування страви.

- Вершини — це завдання: "помити овочі", "нарізати овочі", "зварити суп".
- Ребра-стрілки — це залежності: помити -> нарізати -> зварити. Ця структура є DAG, тому що ви не можете "зварити суп", щоб потім "помити овочі" для нього. Залежності йдуть лише в одному напрямку і не зациклюються.

Він бере цю складну, розгалужену систему залежностей і "витягує" її в одну пряму лінію — правильний покроковий план дій.

Головне правило цього плану: якщо задача 'А' має бути виконана перед задачею 'Б', то в цьому списку 'А' завжди буде стояти раніше за 'Б', що ідеально підходить для будь-якого планування. Наприклад, щоб встановити гру (задача Б), потрібно спочатку встановити драйвери для відеокарти (задача А). Топологічне сортування проаналізує всі такі залежності і видасть вам правильний порядок дій: 1. Встановити драйвери, 2. Встановити гру."

Пошук мостів (Bridge Finding).

Міст — це ребро, видалення якого збільшує кількість компонент зв'язності графа. Простими словами, це критично важливий зв'язок, без якого граф "розпадається" на дві або більше ізольованих частин. Пошук мостів є важливим для аналізу вразливостей у комп'ютерних мережах, транспортних системах та інших мережевих структурах, оскільки вони представляють "єдину точку відмови".

Алгоритм пошуку мостів.

Існує кілька алгоритмів для пошуку мостів. Один з найвідоміших базується на Пошуку в глибину (DFS). Суть його полягає в тому, що під час обходу DFS для кожної вершини відстежується час її "відкриття". Ребро (u, v) (де v є нащадком u у дереві DFS) є мостом, якщо з вершини v або будь-якої з її нащадків немає зворотного шляху до вершини u або будь-якого з її предків. Якщо такий зворотний шлях є, це означає, що навіть після видалення ребра (u, v) зв'язок між ними залишиться через цей обхідний шлях.

- Найкоротша відстань між вершинами (Shortest Path): Це класична задача пошуку шляху з найменшою сумарною вагою ребер. Якщо всі ребра мають однакову вагу, використовується BFS. Для графів з невід'ємною вагою ребер використовується алгоритм Дейкстри. Для графів, де вага може бути від'ємною, застосовується алгоритм Беллмана-Форда.

Пошук найкоротшого шляху в графах

Задача пошуку найкоротшого шляху є однією з найбільш класичних та практично значущих проблем у теорії графів. Від прокладання маршруту в GPS-навігаторі до маршрутизації даних в інтернеті — її вирішення лежить в основі багатьох сучасних технологій.

Суть задачі: Для заданого графа, початкової вершини (джерела) та кінцевої вершини (цілі) знайти шлях, що складається з послідовності ребер, сумарна "вага" яких є мінімальною.

Важливо розуміти, що поняття "найкоротший" залежить від типу графа:

- У незваженому графі найкоротший шлях — це шлях з найменшою кількістю ребер.
- У зваженому графі найкоротший шлях — це шлях з найменшою сумою ваг ребер.

Для вирішення цієї задачі існують різні алгоритми, вибір яких залежить від властивостей графа.

Пошук у ширину (BFS) для незважених графів

Пошук у ширину (Breadth-First Search) — це алгоритм обходу графа, є ідеальним та найпростішим рішенням для знаходження найкоротшого шляху в незважених графах.

- Принцип роботи: BFS досліджує граф "рівень за рівнем". Він починає з початкової вершини, потім відвідує всіх її безпосередніх сусідів (на відстані 1 ребро), потім всіх сусідів цих сусідів (на відстані 2 ребра), і так далі.
- Чому він знаходить найкоротший шлях? Оскільки алгоритм рухається "колами, що розширюються", він гарантовано досягне цільової вершини, пройшовши

найменшу можливу кількість ребер. Будь-який інший шлях за визначенням буде довшим (матиме більше ребер).
- Структура даних: Для своєї роботи використовує чергу (Queue), щоб обробляти вершини в порядку їхнього "відкриття".

Алгоритм Дейкстри — для графів з невід'ємною вагою

Коли ребра мають різну вагу (наприклад, відстань або час), BFS вже не працює, оскільки шлях з меншою кількістю ребер може бути "дорожчим". Тут на допомогу приходить алгоритм Дейкстри.

- Принцип роботи: Це "жадібний" алгоритм. Він підтримує множину відвіданих вершин і для кожної невідвіданої вершини зберігає поточну найкоротшу відому відстань від старту.
    1. На початку відстань до стартової вершини дорівнює 0, до всіх інших — нескінченності.
    2. На кожному кроці алгоритм обирає невідвідану вершину з найменшою поточною відстанню.
    3. Ця вершина позначається як відвідана, і алгоритм "релаксує" ребра, що виходять з неї: він перевіряє, чи не можна покращити (зменшити) шлях до її сусідів, пройшовши через цю щойно відвідану вершину.
    4. Процес повторюється, доки всі вершини не будуть відвідані.
- Структура даних: Для ефективного знаходження вершини з найменшою відстанню використовується черга з пріоритетом (Priority Queue), найчастіше реалізована на основі мін-піраміди (Min-Heap).
- Обмеження: Алгоритм Дейкстри працює коректно лише за умови, що всі ваги ребер є невід'ємними.

Алгоритм Беллмана-Форда — для графів з від'ємною вагою

Що робити, якщо в графі є ребра з від'ємною вагою? (Наприклад, у фінансових задачах це може означати прибуток, а не витрати). Алгоритм Дейкстри в такому випадку може дати неправильний результат, оскільки його "жадібна" логіка (обирати завжди найближчу вершину) тут не спрацює.

- Принцип роботи: Алгоритм Беллмана-Форда є значно повільнішим, але більш надійним. Він працює за принципом ітеративної релаксації:
    1. Як і в Дейкстри, на початку відстань до старту — 0, до інших — нескінченність.
    2. Алгоритм V-1 разів (де V — кількість вершин у графі) проходить по всіх ребрах графа і намагається "релаксувати" кожне з них (тобто, перевіряє, чи не можна покращити шлях через це ребро).
    3. Після V-1 ітерацій гарантовано будуть знайдені всі найкоротші шляхи довжиною до V-1 ребер.
- Виявлення негативних циклів: Алгоритм має унікальну перевагу: він може виявляти цикли з від'ємною сумарною вагою. Якщо після V-1 ітерацій на V-й ітерації все ще можна "релаксувати" хоча б одне ребро, це означає, що в графі існує негативний цикл. У такому циклі можна "намотувати" нескінченно малу відстань, і поняття найкоротшого шляху втрачає сенс.

***Таблиця 5**. Узагальнення порівняння алгоритмів BFS, Дейкстри та Беллмана-Форда*

Алгоритм	Тип графа	Часова складність	Дозволяє від'ємну вагу?
BFS	Незважений	O(V + E)	Ні
Дейкстри	Зважений	O(E log V) (з пірамідою)	Ні
Беллмана-Форда	Зважений	O(V * E)	Так

Задача про максимальний потік. Уявіть мережу труб з різною пропускною здатністю. Задача про максимальний потік (Maximum Flow) полягає у знаходженні максимальної кількості "потоку" (наприклад, води, даних, товарів), який можна передати від однієї вершини (джерела) до іншої (стоку) через цю мережу. Вирішується за допомогою, наприклад, алгоритму Форда-Фалкерсона. Застосовується в логістиці, плануванні мереж та інших оптимізаційних задачах.

На відміну від дерев, що моделюють ієрархію, графи дозволяють представляти складні мережеві системи, де будь-які два вузли можуть бути з'єднані. Граф складається з множини вершин та множини ребер. Здатність моделювати довільні, неієрархічні зв'язки робить графи надзвичайно ефективним інструментом для сучасного ШІ.

Графи знань (Knowledge Graphs) є моделлю представлення даних у вигляді семантичної мережі, де вузли відповідають сутностям реального світу (об'єктам, поняттям), а ребра визначають типи зв'язків між ними.

- Приклад: Для речення "Леонардо да Вінчі, який народився в Італії, написав Мону Лізу" відповідний граф знань міститиме:
  - Вузли: Леонардо да Вінчі, Мона Ліза, Італія.
  - Ребра: написав (від "Леонардо" до "Мона Ліза"), народився в (від "Леонардо" до "Італія").

Графи знань надають системам ШІ здатність до розуміння **контексту** та здійснення логічних висновків. При обробці запиту пошукові системи використовують графи знань для ідентифікації сутностей та зв'язків між ними, що дозволяє надавати більш релевантні відповіді. Цей принцип лежить в основі функціонування інтелектуальних асистентів, систем рекомендацій та семантичного пошуку.

Графові нейронні мережі (Graph Neural Networks - GNNs) є класом нейронних мереж, спеціально розроблених для застосування методів глибокого навчання до даних, структурованих у вигляді графів. Цей напрям є одним із найбільш динамічно розвиваних у сучасному ШІ.

Традиційні архітектури нейронних мереж орієнтовані на обробку даних із регулярною структурою (наприклад, зображення як сітка пікселів, текст як послідовність слів). Проте багато реальних систем, таких як соціальні чи біологічні мережі, не мають такої структури.

GNN функціонують за принципом передачі повідомлень (message passing), у межах якого кожен вузол ітеративно агрегує векторні представлення своїх сусідів для оновлення власного стану. Цей процес дозволяє моделі враховувати як локальні, так і глобальні структурні властивості графа, що є ключовим для вирішення таких завдань:

- Фармацевтика: Прогнозування біохімічної активності молекул на основі їхньої графової структури.
- Аналіз соціальних мереж: Виявлення спільнот, ідентифікація аномальної активності, моделювання поширення інформації.
- Системи рекомендацій: Підвищення якості рекомендацій шляхом аналізу складних взаємозв'язків у графах "користувач-товар".
- Логістика та транспорт: Прогнозування транспортних потоків та оптимізація маршрутів.

Підсумовуючи, можемо сказати, що деревні та графові структури є фундаментальними інструментами для представлення знань та моделювання складних систем у галузі штучного інтелекту.

- Дерева ефективні для моделювання ієрархічних структур та процесів прийняття рішень, забезпечуючи високий рівень інтерпретованості, що є важливим у класичному машинному навчанні.
- Графи дозволяють моделювати складні неієрархічні взаємозв'язки, що є основою для розуміння контексту та роботи з мережевими даними в передових системах ШІ.

Зі зростанням складності завдань, що стоять перед штучним інтелектом — від простої класифікації до аналізу комплексних систем, — роль графових структур, зокрема графових нейронних мереж, продовжує неухильно підвищуватися, відкриваючи нові перспективи для розробки більш досконалих інтелектуальних систем.

2.10. Рекурсія: основа для обходу дерев і графів в Штучному Інтелекті

Дерева та графи є фундаментальними структурами даних для представлення знань у штучному інтелекті. Від дерев рішень, що класифікують дані, до складних графів знань, які лежать в основі пошукових систем та великих мовних моделей, — здатність ефективно переміщатися цими структурами є критично важливою. І найелегантнішим та найпотужнішим інструментом для цього є рекурсія.
Рекурсія — це метод вирішення задачі, за якого функція викликає саму себе для розв'язання менших, ідентичних підзадач. Це не просто техніка програмування, а спосіб мислення, який ідеально відповідає ієрархічній та мережевій природі дерев та графів.
Класичний приклад рекурсії – факторіал.
Перш ніж переходити до складних обходів дерев, давайте розглянемо рекурсію на класичному прикладі — обчисленні факторіала числа n (позначається як n!), що є добутком усіх цілих чисел від 1 до n.

Математично, n! = n * (n-1) * (n-2) * ... * 1. Це відношення можна записати рекурсивно: **n! = n * (n-1)!**. Це і є ключ до рекурсивного рішення. Щоб обчислити факторіал n, нам потрібно знати факторіал n-1.
Будь-яка рекурсивна функція складається з двох частин:

1. Базовий випадок (Base Case): Умова, за якої рекурсія припиняється. Без неї функція викликала б саму себе нескінченно. Для факторіала базовим випадком є 0! = 1.
2. Рекурсивний крок (Recursive Step): Крок, на якому функція викликає саму себе, але зі зміненим аргументом, що наближає її до базового випадку. Для факторіала це n * factorial(n-1).

Приклад на Python:

```
def factorial(n):
    # 1. Базовий випадок: якщо n дорівнює 0, рекурсія зупиняється.
    if n == 0:
        return 1
    # 2. Рекурсивний крок: функція викликає саму себе з меншим аргументом (n-1).
    else:
        return n * factorial(n - 1)

# Обчислимо 5!
# factorial(5) -> 5 * factorial(4)
# factorial(4) -> 4 * factorial(3)
# factorial(3) -> 3 * factorial(2)
# factorial(2) -> 2 * factorial(1)
# factorial(1) -> 1 * factorial(0)
# factorial(0) -> повертає 1 (базовий випадок)
# Результати повертаються назад: 1*1 -> 2*1 -> 3*2 -> 4*6 -> 5*24 = 120

result = factorial(5)
print(f"Факторіал 5! = {result}")
```

Цей простий приклад ідеально демонструє, як рекурсія розбиває складну задачу на простіші, ідентичні підзадачі, поки не досягне найпростішої, рішення для якої відоме. Причина, чому рекурсія настільки природна для дерев та графів, криється в їхній самоподібній (фрактальній) структурі.

- Дерево складається з кореневого вузла та піддерев, кожне з яких, у свою чергу, є повноцінним деревом. Обробка всього дерева зводиться до обробки одного вузла і подальшої обробки його нащадків, що є меншими копіями вихідної задачі.
- Граф складається з вузлів, і якщо розглянути будь-який вузол та його сусідів, то від кожного сусіда можна почати обхід решти графа, що є тією ж самою задачею.

Рекурсивний підхід дозволяє нам написати просту функцію, яка знає, як обробити один вузол, і довіряє самій собі обробити всіх його нащадків чи сусідів. Це значно спрощує код, роблячи його більш читабельним у порівнянні з ітеративними підходами. Ітеративний підхід вимагав би від програміста вручну керувати стеком для відстеження вузлів, які потрібно відвідати, тоді як рекурсія робить це автоматично за допомогою системного стека викликів функцій.

Рекурсивний обхід Дерева (Пошук у глибину - DFS)

Пошук у глибину (Depth-First Search) — це класичний алгоритм обходу, який досліджує гілки дерева настільки глибоко, наскільки це можливо, перш ніж повернутися назад. Рекурсія робить його реалізацію тривіальною.

Алгоритм:

1. Обробити поточний вузол (наприклад, вивести його значення).
2. Для кожного нащадка поточного вузла рекурсивно викликати ту ж саму функцію обходу.

Приклад на Python:

```
# Клас для представлення вузла дерева
class Node:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children if children is not None else []

# Рекурсивна функція обходу
def traverse_tree(node):
    # 1. Обробляємо поточний вузол до того, як відвідати його нащадків (це називається прямим порядком обходу або pre-order traversal).
    print(f"Відвідуємо вузол: {node.value}")

    # 2. Базовий випадок (неявний): якщо нащадків немає, цикл не виконається, і рекурсія завершиться
    # 3. Рекурсивний крок: викликаємо функцію для кожного нащадка
    for child in node.children:
        traverse_tree(child)

    # Можна обробляти вузол і після відвідування всіх нащадків (post-order traversal)
    # print(f"Повертаємося з вузла: {node.value}")

# Створимо просте дерево:
#      1
#     / \
#    2   3
#   / \   \
#  5   6   4
tree = Node(1, [
    Node(2, [
        Node(5),
        Node(6)
    ]),
    Node(3, [
        Node(4)
    ])
])

print("--- Починаємо обхід дерева в глибину ---")
traverse_tree(tree)
```

Як бачимо, код є лаконічним та інтуїтивно зрозумілим. Він точно відображає ієрархічну природу дерева. Комп'ютер "пам'ятає", куди повернутися, використовуючи стек викликів функцій.

Рекурсивний обхід графа: виклики та рішення

Обхід графа схожий на обхід дерева, але має одну критичну відмінність: графи можуть містити цикли. Якщо просто застосувати той самий алгоритм, що й для дерева, ми ризикуємо потрапити в нескінченну рекурсію (наприклад, A -> B -> A -> B -> ...).
Рішення наступне. Потрібно відстежувати вже відвідані вузли. Для цього ідеально підходить множина (set) через її високу швидкість перевірки наявності елемента (в середньому O(1)). Використання списку (O(n)) для цієї мети зробило б алгоритм значно повільнішим на великих графах.
Алгоритм (DFS для графа):

1. Перевірити, чи був поточний вузол відвіданий раніше. Якщо так, негайно завершити поточний виклик.
2. Якщо ні, позначити поточний вузол як відвіданий, додавши його до множини.
3. Обробити поточний вузол.
4. Для кожного сусіда поточного вузла рекурсивно викликати функцію обходу.

```
# Представимо граф за допомогою списків суміжності (словник)
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Рекурсивна функція обходу графа
def traverse_graph(node, visited):
    # 1. Перевірка на цикл - базовий випадок
    if node in visited:
        return

    # 2. Позначаємо вузол як відвіданий
    visited.add(node)

    # 3. Обробляємо поточний вузол
    print(f"Відвідуємо вузол: {node}")

    # 4. Рекурсивний крок для сусідів
    for neighbor in graph[node]:
        traverse_graph(neighbor, visited)

print("\n--- Починаємо обхід графа в глибину ---")
visited_nodes = set() # Множина для зберігання відвіданих вузлів
traverse_graph('A', visited_nodes)
```

Рекурсивний обхід лежить в основі багатьох алгоритмів штучного інтелекту.

1. Ігровий ШІ (Алгоритм Minimax)
У таких іграх, як шахи чи хрестики-нулики, алгоритм Minimax будує дерево можливих ходів.

- Кожен вузол — це стан гри.
- Рекурсивна функція оцінює, наскільки "хорошим" є поточний стан.
- Вона викликає себе для всіх можливих наступних ходів, припускаючи, що опонент завжди робитиме найкращий для себе хід (мінімізуючи нашу вигоду).
- Результати з глибини рекурсивно піднімаються вгору: функція на рівні 'гравець' обирає хід з максимальною оцінкою (Max), а на рівні 'опонент' — з мінімальною (Min). Це дозволяє знайти оптимальний хід у поточному стані, що максимізує шанси на перемогу.

2. Обробка природної мови (NLP)
Синтаксичний аналіз речень часто зводиться до побудови дерева синтаксичного розбору. Речення розбивається на фрази, які, у свою чергу, розбиваються на менші фрази або слова. Рекурсивні функції можуть обходити це дерево для аналізу граматичної структури, визначення семантичних зв'язків (хто що зробив), виділення ключових сутностей або перекладу. Наприклад, щоб зрозуміти сенс складного речення, потрібно рекурсивно зрозуміти сенс його підрядних частин.

3. Графи знань та логічне виведення
Пошук відповіді на питання в базі знань (наприклад, "хто є дідусем Олени?") — це задача пошуку шляху в графі знань. Рекурсивний пошук у глибину може ефективно досліджувати зв'язки між сутностями (Олена -> має_батька -> X -> має_батька -> Y), щоб знайти відповідь. Функція може рекурсивно шукати шлях від Олени за відношенням має_батька, а потім від знайденого батька шукати його батька.

4. Планування та розв'язання задач
Багато задач у ШІ, від пошуку маршруту до планування дій робота, можна представити як пошук у просторі станів, який, по суті, є графом. Кожен стан — це вузол, а можливі дії — це ребра. Рекурсивні алгоритми, такі як DFS, є основою для дослідження цього простору та знаходження послідовності дій, що веде до мети. Наприклад, робот-складальник може рекурсивно оцінювати послідовності дій: 'Якщо я спочатку візьму деталь А, чи зможу я потім приєднати деталь Б?'. Так він досліджує дерево можливих станів, відкидаючи тупикові гілки, поки не знайде правильну послідовність для збирання пристрою.
Отже, рекурсія — це не просто математична концепція, а потужний практичний інструмент, що дозволяє елегантно та ефективно вирішувати задачі, пов'язані з ієрархічними та мережевими даними. Для штучного інтелекту, де дерева та графи використовуються для моделювання знань, прийняття рішень та пошуку, глибоке розуміння рекурсивного мислення є фундаментальним. Навіть у сучасних складних архітектурах, таких як графові нейронні мережі, ідеї рекурсивного оновлення та поширення інформації залишаються центральними.

2.11. Модулі в Python: від основ до інструментів ШІ

Однією з найбільших переваг мови програмування Python є її величезна екосистема модулів та бібліотек. Саме вони перетворюють Python з мови загального призначення на потужний інструмент для вирішення спеціалізованих завдань, особливо у сфері

штучного інтелекту. Ці інструменти надають готові, оптимізовані рішення для складних математичних обчислень, маніпуляцій з даними та побудови моделей, дозволяючи розробникам та дослідникам зосередитись на вирішенні самої проблеми, а не на написанні базового функціоналу.

Модуль — це файл з розширенням .py, який містить визначення та інструкції Python. Це може бути набір функцій, класів або змінних, які можна легко використовувати в інших програмах. Коли говорять про бібліотеку або пакет, зазвичай мають на увазі набір пов'язаних модулів.

Основні переваги використання модулів:

- Організація коду - модулі дозволяють розбити велику програму на менші, логічно пов'язані частини, що значно спрощує її розробку, тестування та підтримку.
- Повторне використання - ви можете написати корисну функцію один раз і використовувати її в багатьох проєктах, просто імпортувавши відповідний модуль.
- Розширення функціоналу - вам не потрібно писати все з нуля. Існує величезна кількість готових модулів для вирішення практично будь-яких завдань, від математичних обчислень до створення веб-серверів та нейронних мереж.

Для використання функціоналу з модуля використовується інструкція import.

```
# Спосіб 1: Імпортувати весь модуль
import math
print(math.sqrt(16)) # Потрібно вказувати назву модуля

# Спосіб 2: Імпортувати конкретну функцію
from math import sqrt
print(sqrt(16)) # Можна викликати функцію напряму

# Спосіб 3: Використання псевдоніма (аліаса) - дуже поширена практика
import numpy as np
# Тепер можна звертатися до numpy, використовуючи коротке ім'я np
```

Екосистема ШІ в Python є надзвичайно розвиненою. Ось перелік основних модулів, згрупованих за типовими етапами роботи над ШІ-проєктом.

1. Робота з даними та наукові обчислення

Це фундамент, на якому будується будь-яка модель машинного навчання.

- NumPy (Numerical Python) - основа для всіх наукових обчислень у Python. Надає потужний об'єкт — багатовимірний масив (ndarray), та оптимізовані функції для математичних операцій над цими масивами. Робота з NumPy набагато швидша, ніж зі стандартними списками Python, оскільки операції виконуються на низькорівневому, компільованому коді (C або Fortran), а не на рівні інтерпретатора Python; використовується завжди, коли ви працюєте з числовими даними, матрицями, векторами. Це залежність №1 для більшості інших бібліотек.

  Приклад:
  ```
  import numpy as np
  my_array = np.array([1, 2, 3, 4, 5])
  ```

```
# Векторизована операція - застосовується до всього масиву одразу
doubled_array = my_array * 2
print(doubled_array) # Виведе: [ 2  4  6  8 10]
```

- Pandas - надає зручні структури даних (зокрема, DataFrame, що є аналогом таблиці з іменованими стовпцями та індексами) та інструменти для маніпуляції та аналізу даних. Дозволяє легко читати та записувати дані з різних форматів (CSV, Excel, бази даних); використовується для очищення, трансформації, фільтрації, групування та аналізу табличних даних.

```
Приклад:
import pandas as pd
data = {'Ім\'я': ['Олена', 'Іван', 'Марія'], 'Вік': [28, 34, 29]}
df = pd.DataFrame(data)
# Фільтрація даних
adults = df[df['Вік'] > 30]
print(adults)
```

2. Класичне машинне навчання (ML)

- Scikit-learn - це "швейцарський ніж" для класичного машинного навчання. Ця бібліотека містить прості та ефективні інструменти для вирішення завдань класифікації, регресії, кластеризації, а також для підготовки даних (масштабування, кодування) та оцінки моделей; використовується, коли вам потрібно швидко застосувати стандартні алгоритми, такі як лінійна регресія, дерева рішень, метод опорних векторів (SVM) або K-середніх. Її перевага — єдиний та послідовний API (.fit(), .predict(), .transform()).

```
Приклад:
from sklearn.linear_model import LinearRegression
# У нас є дані: X - розмір будинку, y - ціна
X = [[100], [150], [200], [250]]
y = [120, 170, 230, 280]
model = LinearRegression()
model.fit(X, y) # Навчаємо модель
prediction = model.predict([[175]]) # Прогнозуємо ціну
print(f"Прогнозована ціна: {prediction[0]}")
```

3. Глибоке навчання (Deep Learning)

Це модулі для побудови та навчання нейронних мереж.

- TensorFlow - комплексна платформа для глибокого навчання, розроблена Google. Вона чудово підходить для створення масштабних моделей та їх розгортання в промислових умовах. TensorFlow має потужну екосистему інструментів, включаючи TensorBoard для візуалізації, TensorFlow Serving для розгортання моделей та TensorFlow Lite для мобільних пристроїв.
- PyTorch - платформа для глибокого навчання, розроблена Meta. Вона стала надзвичайно популярною в дослідницькому середовищі завдяки своїй гнучкості та інтуїтивно зрозумілому, "пайтонічному" синтаксису. PyTorch використовує динамічні обчислювальні графи (Eager Execution), що спрощує дебагінг та роботу зі

складними архітектурами, де структура мережі може змінюватися під час виконання.

- Keras - високорівневий API, який робить процес побудови нейронних мереж максимально простим та швидким. Keras може працювати поверх TensorFlow (що є стандартною конфігурацією сьогодні) або інших бекендів. Keras - ідеальний вибір для початківців та для швидкого прототипування моделей завдяки своєму мінімалістичному та модульному дизайну.

Приклад:

```
# from tensorflow import keras
# from tensorflow.keras import layers
# model = keras.Sequential([
#     layers.Dense(64, activation="relu", input_shape=(784,)),
#     layers.Dense(10, activation="softmax")
# ])
```

4. Обробка природної мови (NLP)

- NLTK (Natural Language Toolkit) - одна з найстаріших і найповніших бібліотек для NLP. Чудово підходить для освітніх цілей та для виконання базових завдань, таких як токенізація, стемінг, лематизація, POS-тегування.
- Hugging Face Transformers - ця бібліотека здійснила справжню революцію в NLP, демократизувавши доступ до найсучасніших трансформерних архітектур (як-от BERT, GPT, T5). Вона надає уніфікований API для завантаження тисяч попередньо навчених моделей з центрального репозиторію (Hugging Face Hub) та їх легкого застосування для вирішення різноманітних завдань: від класифікації тексту та відповідей на питання до перекладу та генерації тексту.

Приклад:

```
# from transformers import pipeline
# classifier = pipeline("sentiment-analysis")
# result = classifier("Я люблю цю бібліотеку!")
# print(result) # [{'label': 'POSITIVE', 'score': ...}]
```

5. Візуалізація даних

- Matplotlib - фундаментальна бібліотека для створення статичних, анімованих та інтерактивних візуалізацій. Дозволяє створювати широкий спектр графіків: лінійні, стовпчикові, гістограми тощо. Надає повний контроль над кожним елементом графіка.
- Seaborn - модуль, побудований на основі Matplotlib, ця бібліотека дозволяє створювати більш привабливі та інформативні статистичні графіки (як-от теплові карти, скрипкові діаграми) за допомогою меншої кількості коду.

***Таблиця 6**. Найпоширеніші вбудовані модулі*

Модуль	Основне призначення	Приклади використання
math	Надає доступ до математичних функцій та констант, що виходять за межі базової арифметики.	math.sqrt(16) (квадратний корінь), math.sin(x) (синус), math.pi (число π), math.log(x) (логарифм).

random	Дозволяє генерувати псевдовипадкові числа та робити випадковий вибір.	random.randint(1, 10) (випадкове ціле число), random.random() (випадкове число від 0 до 1), random.choice(my_list) (випадковий елемент зі списку).
os	Надає інструменти для взаємодії з операційною системою.	os.path.exists('file.txt') (перевірка існування файлу), os.makedirs('folder') (створення папки), os.listdir('.') (отримання списку файлів у директорії).
sys	Надає доступ до параметрів та функцій, специфічних для інтерпретатора Python.	sys.argv (аргументи командного рядка), sys.platform (інформація про ОС), sys.exit() (завершення роботи програми).
datetime	Дозволяє працювати з датами та часом.	datetime.datetime.now() (поточна дата і час), datetime.date(2025, 1, 1) (створення об'єкта дати), timedelta (обчислення різниці в часі).
json	Надає інструменти для роботи з даними у форматі JSON (JavaScript Object Notation).	json.loads(json_string) (перетворення рядка JSON на словник Python), json.dumps(my_dict) (перетворення словника на рядок JSON).
collections	Надає спеціалізовані типи даних-контейнерів.	collections.deque (двостороння черга), collections.Counter (словник-лічильник), collections.defaultdict (словник зі значенням за замовчуванням).
re	Надає інструменти для роботи з **р**егулярними **в**иразами (Regular Expressions) для пошуку та маніпуляції текстовими шаблонами.	re.search(pattern, text) (пошук шаблону), re.findall(pattern, text) (пошук усіх входжень), re.sub(pattern, repl, text) (заміна).

Найпоширеніші сторонні бібліотеки (особливо для ШІ)

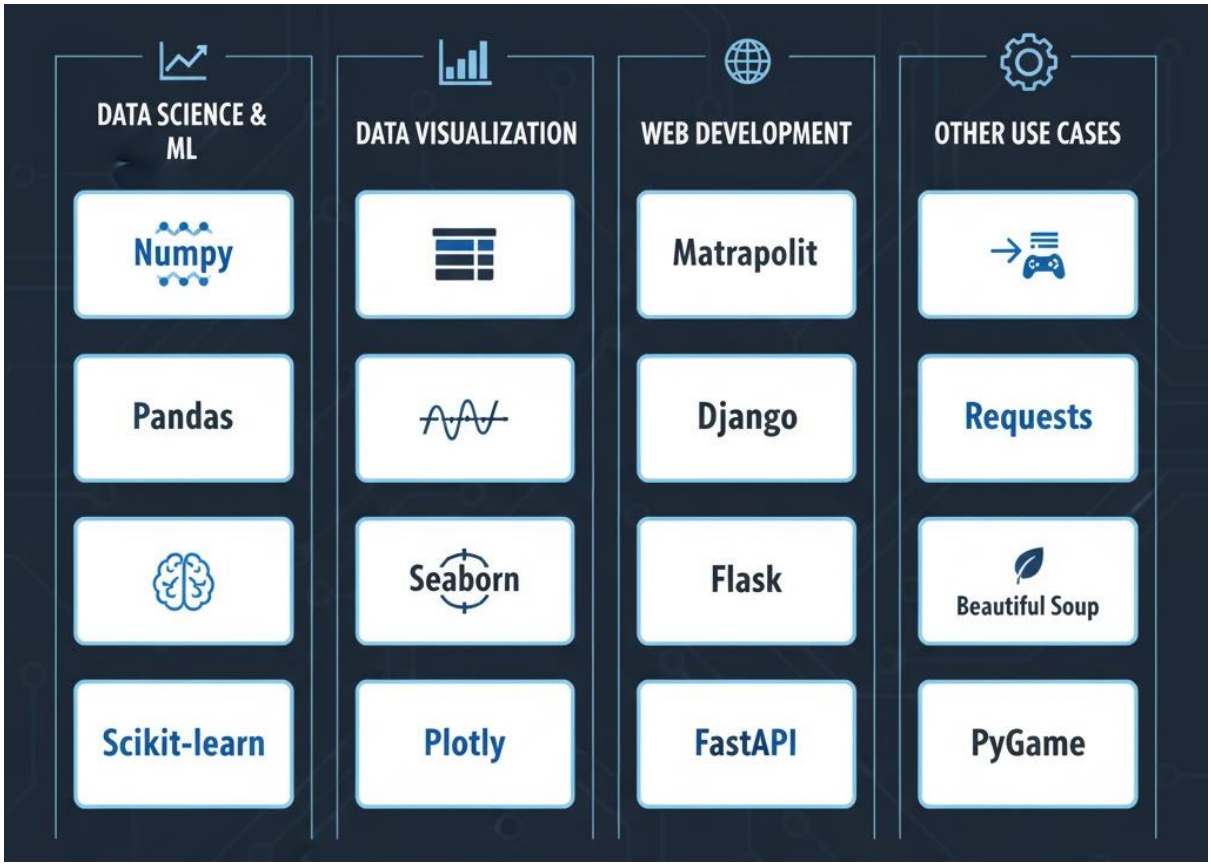


Рис.14. Найпоширеніші сторонні бібліотеки (особливо для ШІ)

На відміну від вбудованих модулів, сторонні бібліотеки потрібно встановлювати окремо за допомогою менеджера пакетів, такого як pip. Саме ці бібліотеки перетворили Python на мову №1 для наукових обчислень, аналізу даних та штучного інтелекту. Ось деякі з найважливіших.

***Таблиця 7**. Бібліотеки, призначення та приклади використання*

Бібліотека (імпорт)	**Основне призначення**	**Приклади використання**
NumPy (np)	Фундаментальна бібліотека для наукових обчислень. Надає потужні багатовимірні масиви (ndarray) та оптимізовані математичні функції.	np.array([1, 2, 3]), np.dot(a, b) (скалярний добуток), np.mean(arr) (середнє значення).
Pandas (pd)	Інструмент для маніпуляції та аналізу табличних даних. Надає структури DataFrame та Series.	pd.read_csv('data.csv'), df['column'].value_counts(), df.groupby('group').mean().
Matplotlib (plt)	Основоположна бібліотека для створення статичних, анімованих та інтерактивних візуалізацій та графіків.	plt.plot(x, y), plt.scatter(x, y), plt.title('Назва'), plt.show().
Scikit-learn (sklearn)	"Швейцарський ніж" для класичного машинного навчання. Містить	from sklearn.linear_model import LogisticRegression,

	алгоритми класифікації, регресії, кластеризації.	model.fit(X, y), model.predict(X_new).
TensorFlow (tf) / **PyTorch** (torch)	Потужні фреймворки для глибокого навчання (Deep Learning). Надають тензори з підтримкою GPU та автоматичного диференціювання.	Побудова та тренування нейронних мереж. torch.Tensor, tf.keras.layers.Dense.
Hugging Face Transformers	Надає доступ до тисяч попередньо навчених трансформерних моделей (BERT, GPT) для задач NLP.	pipeline('sentiment-analysis'), AutoModel.from_pretrained('bert-base-uncased').
Seaborn (sns)	Високорівнева бібліотека для створення привабливих статистичних графіків, побудована на основі Matplotlib.	sns.heatmap(data), sns.violinplot(x, y), sns.pairplot(df).

Отже, для старту у сфері ШІ з використанням Python варто зосередитися на вивченні "великої трійки" для роботи з даними та класичного ML: NumPy, Pandas та Scikit-learn. Вони є основою для більшості завдань. Після цього, залежно від ваших інтересів, можна заглиблюватися у фреймворки глибокого навчання, такі як PyTorch або TensorFlow/Keras, для створення складних нейронних мереж, або в спеціалізовані бібліотеки, як-от Hugging Face Transformers, для роботи з текстом. Модульна структура Python дозволяє комбінувати ці інструменти, створюючи потужні та гнучкі рішення для найскладніших завдань штучного інтелекту.

2.12. Робота з файлами в Python: Основи та нюанси

Робота з файлами є фундаментальною навичкою в програмуванні. Програми використовують файли для зберігання даних, конфігурацій, логів, результатів роботи та для обміну інформацією з іншими програмами. Python надає простий та потужний інструментарій для взаємодії з файловою системою.

Основа основ - функція open()

Все починається з вбудованої функції open(), яка відкриває файл і повертає файловий об'єкт.

Основний синтаксис:
open(file_path, mode='r', encoding=None)

- file_path - рядок, що містить шлях до файлу (наприклад, 'data/my_file.txt').
- mode - режим, у якому відкривається файл. Це один з найважливіших параметрів.
- encoding - кодування файлу, це дуже важливий нюанс, про який поговоримо окремо.

***Таблиця 8**. Режими доступу до файлу*

Режим	Опис
'r'	**Read (читання)**. Режим за замовчуванням. Викликає помилку, якщо файл не існує.
'w'	**Write (запис)**. Створює новий файл для запису. Якщо файл вже існує, його вміст буде стерто!
'a'	**Append (додавання)**. Відкриває файл для дозапису в кінець. Якщо файл не існує, він буде створений.
'b'	**Binary (бінарний режим)**. Додається до інших режимів (наприклад, 'rb', 'wb') для роботи з нетекстовими файлами (зображення, аудіо).
'+'	**Read/Write (читання та запис)**. Додається до інших режимів (наприклад, 'r+', 'w+') для одночасного читання та запису.

Контекстний менеджер with

Хоча можна відкрити файл так: f = open(...) і закрити його вручну f.close(), це не є рекомендованим підходом. Якщо під час роботи з файлом станеться помилка, метод .close() може не викликатись, і файл залишиться відкритим, що може призвести до проблем.

Правильний, сучасний спосіб — використовувати конструкцію with open(...). Вона гарантує, що файл буде автоматично закрито після завершення блоку, навіть якщо виникнуть помилки.

```
try:
    with open('my_document.txt', 'w', encoding='utf-8') as f:
        f.write('Це надійний спосіб роботи з файлами.\n')
        # Файл автоматично закриється тут
except IOError as e:
    print(f"Сталася помилка вводу-виводу: {e}")
```

Основні команди та операції

1. Читання з файлу

Існує кілька способів прочитати вміст файлу.

- .read(size): Читає вказану кількість байтів (або весь файл, якщо size не вказано).
  - Нюанс: Будьте обережні, викликаючи .read() без аргументів на великих файлах, оскільки це може вичерпати пам'ять.
- .readline(): Читає один рядок з файлу, включаючи символ нового рядка \n.
- .readlines(): Читає всі рядки файлу і повертає їх у вигляді списку. Також може споживати багато пам'яті.
- Ітерація по файлу (найкращий спосіб): Найбільш "пайтонічний" та ефективний з точки зору пам'яті спосіб читати файл рядок за рядком.

Приклад.

```
# Створимо тестовий файл
with open('shopping_list.txt', 'w', encoding='utf-8') as f:
    f.write('Молоко\n')
    f.write('Хліб\n')
    f.write('Яйця\n')

# Читаємо та виводимо його вміст
print("--- Читаємо файл рядок за рядком ---")
try:
    with open('shopping_list.txt', 'r', encoding='utf-8') as f:
        for line in f:
            print(line.strip()) # .strip() видаляє зайві пробіли та символи нового рядка
('\n') з початку та кінця рядка.
except FileNotFoundError:
    print("Файл не знайдено!")
```

2. Запис у файл

- .write(string): Записує вказаний рядок у файл. Не додає символ нового рядка автоматично — його потрібно додавати вручну ('\n').
- .writelines(list_of_strings): Записує список рядків у файл. Також не додає символи нового рядка.

Приклад.

```
lines_to_write = ['Перший рядок\n', 'Другий рядок\n', 'Третій рядок\n']

with open('output.txt', 'w', encoding='utf-8') as f:
    f.writelines(lines_to_write) # Важливо: .writelines() не додає символи нового
рядка автоматично.
```

3. Додавання в кінець файлу

Режим 'a' (append) дозволяє додавати дані, не стираючи існуючий вміст.

```
with open('output.txt', 'a', encoding='utf-8') as f:
    f.write('Це доданий рядок.\n')
```

У контексті штучного інтелекту навички роботи з файлами виходять на новий рівень важливості. Весь життєвий цикл ШІ-проєкту — від збору даних до розгортання моделі — нерозривно пов'язаний з операціями читання та запису. Розглянемо ключові аспекти:

1. *Завантаження датасетів та конфігурацій.* Моделі ШІ навчаються на даних, які майже завжди зберігаються у файлах (CSV, JSON, текстові файли). Бібліотеки, як-от Pandas, використовують вбудовані функції Python "під капотом" для читання цих файлів (`pd.read_csv('data.csv')`). Аналогічно, конфігураційні файли (наприклад, JSON), що містять гіперпараметри моделі, читаються для налаштування експериментів.
2. *Робота з великими даними (Big Data).* Датасети для ШІ можуть бути гігантськими і не вміщатися в оперативну пам'ять. У цьому випадку построкове читання файлу стає єдиним можливим варіантом. Саме за таким принципом працюють генератори даних у TensorFlow та PyTorch: вони читають дані з файлу

не всі одразу, а невеликими партіями (batches), що дозволяє навчати моделі на даних будь-якого розміру.

3. *Збереження та завантаження моделей.* Навчена модель (її архітектура та мільйони параметрів-ваг) — це цінний актив. Щоб не навчати її заново щоразу, її зберігають у файл. Зазвичай для цього використовується бінарний режим запису (`'wb'`), оскільки ваги моделі зберігаються не як текст, а як компактний потік байтів. Потім модель можна легко завантажити (`'rb'`) для подальшого використання чи донавчання.
4. *Логування та відтворюваність експериментів.* Процес навчання моделі може тривати годинами або днями. Вся важлива інформація (значення помилки на кожній епосі, метрики точності) записується у лог-файли (часто в режимі додавання `'a'`). Це дозволяє аналізувати процес та відстежувати прогрес. Збереження логів та конфігурацій робить результати наукових досліджень та комерційних проєктів відтворюваними.

4. Робота з файловою системою: os та pathlib

Для маніпуляцій із самими файлами (перевірка існування, видалення, створення папок) використовуються модулі os та pathlib.

- os.path.exists(path): Перевіряє, чи існує файл або папка.
- os.remove(path): Видаляє файл.
- os.makedirs(path): Створює папку.
- pathlib.Path('folder'): Сучасний, об'єктно-орієнтований підхід, що робить код чистішим.

Приклад.

```
import os
import pathlib

# Класичний підхід з os
if not os.path.exists('my_folder'):
    os.makedirs('my_folder')

# Сучасний підхід з pathlib
p = pathlib.Path('my_new_folder')
p.mkdir(exist_ok=True) # Створює папку, не викликаючи помилки, якщо вона вже існує
```

4. Абсолютні та відносні шляхи

- Відносний шлях ('data/file.txt') вказується відносно поточної робочої директорії вашого скрипта.
- Абсолютний шлях ('C:/Users/User/Documents/file.txt' для Windows або '/home/user/documents/file.txt' для Linux/macOS) вказує повний шлях від кореня файлової системи.

Для створення крос-платформенних шляхів краще використовувати os.path.join() або об'єкти pathlib, які автоматично використовують правильний роздільник (\ або /).

Приклад.

```
import os
path = os.path.join('data', 'subfolder', 'my_file.txt')
print(f"Сконструйований шлях: {path}") # На Windows буде
'data\\subfolder\\my_file.txt'
```

Отже, робота з файлами в Python є простою та логічною, якщо дотримуватися кількох ключових правил:

1. Завжди використовуйте менеджер контексту with open(...) для автоматичного закриття файлів.
2. Завжди явно вказуйте кодування, переважно encoding='utf-8'.
3. Пам'ятайте про різницю між режимами 'r', 'w' та 'a', щоб випадково не стерти важливі дані.
4. Для маніпуляцій із файловою системою використовуйте сучасний модуль pathlib або класичний os.

Дотримання цих принципів зробить ваш код надійним, портативним та легким для розуміння

2.13. Алгоритми опрацювання змінних без індексів в Python

Традиційні структури даних, такі як списки (lists) або кортежі (tuples), організовані за допомогою числових індексів (0, 1, 2, ...). Це зручно, коли порядок елементів є важливим. Однак існує цілий клас задач, де порядок не має значення, а доступ за числовим індексом є неефективним або нелогічним. Python пропонує потужні інструменти для опрацювання таких даних.

Коли відмова від індексів є перевагою?

- Коли потрібно швидко перевірити, чи існує елемент у колекції, або гарантувати унікальність елементів.
- Асоціативні дані: Коли дані логічно пов'язані у пари "ключ-значення", де ключ є унікальним ідентифікатором (наприклад, ім'я користувача, ID продукту).
- Обробка потокових даних: Коли дані надходять послідовно (наприклад, з файлу або мережі) і їх неможливо чи недоцільно завантажувати в пам'ять повністю.

1. Множини (Sets): Робота з унікальними елементами

Множина — це невпорядкована колекція *унікальних* елементів. Це ідеальна структура даних, коли вам потрібно зберігати елементи без дублікатів і швидко перевіряти їх наявність.

- Основна ідея: Множини реалізовані на основі хеш-таблиць, що забезпечує надзвичайно швидке виконання операцій додавання, видалення та перевірки належності (в середньому за час O(1)).
- Створення: my_set = {1, 2, 3, "a"} або my_set = set([1, 2, 2, 3]). В другому випадку дублікат 2 буде автоматично видалено.

Алгоритми опрацювання:

а) Перевірка наявності та додавання

Замість перебору всього списку за допомогою циклу for, ми використовуємо оператор in.

```
permissions = {"read", "write", "execute"}

# Перевірка (дуже швидко)
if "read" in permissions:
    print("Користувач має право на читання.")

# Додавання елемента
permissions.add("delete")
print(permissions)  # Порядок виводу може бути довільним

# Спроба додати існуючий елемент нічого не змінить
permissions.add("write")
print(permissions)
```

б) Операції над множинами

Множини дозволяють виконувати потужні математичні операції, що є основою для багатьох алгоритмів фільтрації та порівняння.

- Об'єднання (| або .union()): Усі елементи з обох множин.
- Перетин (& або .intersection()): Тільки спільні елементи.
- Різниця (- або .difference()): Елементи, що є в першій множині, але відсутні в другій.

Приклад: Аналіз навичок кандидатів.

```
candidate1_skills = {"python", "git", "sql", "docker"}
candidate2_skills = {"java", "git", "spring", "sql"}

# Які навички є хоча б в одного з них?
all_skills = candidate1_skills | candidate2_skills
print(f"Загальний пул навичок: {all_skills}")

# Які навички є спільними?
common_skills = candidate1_skills & candidate2_skills
print(f"Спільні навички: {common_skills}")

# Які навички є у першого кандидата, але немає у другого?
unique_to_c1 = candidate1_skills - candidate2_skills
print(f"Унікальні навички першого кандидата: {unique_to_c1}")
```

2. Словники (Dictionaries): Доступ за ключем

Словник — це колекція пар "ключ-значення". Доступ до значення відбувається не за числовим індексом, а за його унікальним ключем.

- Основна ідея: Як і множини, словники використовують хеш-таблиці для швидкого доступу за ключем (в середньому O(1)). Це набагато ефективніше, ніж шукати елемент у списку за певним полем.
- Створення: user = {"name": "Іван", "age": 30, "city": "Київ"}.

Алгоритми опрацювання:

Ітерація по словнику відбувається не по індексах, а по його компонентах: ключах, значеннях або парах.

```
user_profile = {
    "username": "coder123",
    "email": "coder@example.com",
    "is_active": True,
    "points": 150
}

# Ітерація по ключах (стандартна поведінка)
print("\nКлючі профілю:")
for key in user_profile:
    print(key)

# Ітерація по значеннях
print("\nЗначення профілю:")
for value in user_profile.values():
    print(value)

# Ітерація по парах "ключ-значення" - найпоширеніший варіант
print("\nПовний профіль:")
for key, value in user_profile.items():
    print(f"{key.capitalize()}: {value}")
```

3. Ітератори та Генератори: Послідовний доступ

Іноді дані настільки великі, що їх не можна завантажити в пам'ять (наприклад, великий текстовий файл), або вони генеруються "на льоту". У таких випадках використовують ітератори.

Ітератор — це об'єкт, який дозволяє отримувати елементи послідовно, один за одним. Цикл for у Python "під капотом" працює саме з ітераторами.

Приклад: Обробка файлу рядок за рядком без завантаження всього файлу.

```
# Уявімо, що 'large_log.txt' має мільйони рядків
# Відкриття файлу створює ітератор, який читає файл рядок за рядком

try:
    with open('large_log.txt', 'r', encoding='utf-8') as f:
        error_count = 0
        for line in f:  # 'line' отримується послідовно, без індексів
            if "ERROR" in line:
                error_count += 1
    print(f"Знайдено {error_count} помилок у лог-файлі.")

except FileNotFoundError:
    print("Файл 'large_log.txt' не знайдено. Приклад не може бути виконаний.")
```

Цей підхід є надзвичайно ефективним з точки зору використання пам'яті, оскільки в будь-який момент часу в пам'яті знаходиться лише один рядок.

Отже, опрацювання змінних без індексів є не просто альтернативою, а потужною парадигмою в сучасному програмуванні на Python. Множини (set) незамінні для роботи з унікальними даними та виконання логічних операцій. Словники (dict) дозволяють

будувати складні структури з доступом за змістовним ключем, що робить код більш читабельним. Ітератори дають можливість ефективно обробляти великі обсяги даних, абстрагуючись від їх зберігання. Використання цих підходів дозволяє писати більш швидкий, чистий та масштабований код.

Контрольні запитання до розділу

1. Ядро маніпуляції даними. Які основні ролі відіграють бібліотеки NumPy та Pandas у типовому робочому процесі ШІ/МН, і чому використання стандартних списків Python для великомасштабних числових обчислень є, як правило, неефективним?

2. Уніфікований API Scikit-learn. Бібліотеку Scikit-learn часто хвалять за її послідовний та уніфікований API. Назвіть три основні методи, які є спільними для більшості її об'єктів (моделей, трансформерів), і коротко поясніть призначення кожного з них.

3. Фреймворки для глибокого навчання. У чому полягає ключова філософська та практична різниця між підходом PyTorch з його "динамічними обчислювальними графами" (Eager Execution) та традиційним підходом TensorFlow "визнач-і-запусти" (статичні графи)? Який з них може бути кращим для дослідницьких цілей і чому?

4. Рівні абстракції. Як бібліотека Keras співвідноситься з TensorFlow? В якому сценарії розробник обрав би використання високорівневого API Keras напряму, а в якому — звернувся б до низькорівневих функціональностей TensorFlow?

5. Сучасний NLP. Бібліотека Hugging Face Transformers здійснила революцію в обробці природної мови. У чому її головна перевага над старішими бібліотеками, як-от NLTK, особливо при вирішенні складних завдань, наприклад, генерації тексту чи відповідей на питання?

6. Вибір правильного інструменту. Уявіть, що вам потрібно створити модель для прогнозування цін на нерухомість на основі CSV-файлу, що містить такі ознаки, як площа, кількість кімнат та розташування. Які три модулі з обговорених стали б вашим основним набором інструментів, і яку роль кожен з них відіграв би в цьому проєкті?

7. Роль візуалізації. Чому візуалізація даних (з використанням Matplotlib/Seaborn) вважається критично важливим етапом як *до* початку навчання моделі, так і *після* його завершення? Наведіть по одному прикладу для кожного випадку.

8. Синергія модулів. Поясніть, як NumPy та Pandas працюють разом. Яким чином структура DataFrame в Pandas використовує потужність масивів NumPy "під капотом"?

9. "Навіщо" існують модулі? Чому, як правило, є поганою ідеєю намагатися реалізувати складний алгоритм, такий як метод опорних векторів (SVM) або нейронну мережу, з нуля для промислового застосування, замість того, щоб використовувати готові бібліотеки, як-от Scikit-learn або PyTorch?

10. Рефлексія та наступні кроки. Базуючись на обговорених модулях (NumPy, Pandas, Scikit-learn, PyTorch/TensorFlow, Transformers), яка сфера (наприклад, аналіз даних, класичне МН, глибоке навчання, NLP) здається вам найбільш цікавою, і вивченню якого модуля ви б надали пріоритет для подальшого розвитку в цьому напрямку?

Тести для самоконтролю

1. Яка ключова відмінність між списком (list) та кортежем (tuple) в Python?
    a) Списки можуть зберігати елементи різних типів, а кортежі — ні.
    b) Доступ до елементів у кортежах швидший, ніж у списках.
    c) Списки є змінними (mutable), тобто їх можна змінювати, а кортежі — незмінними (immutable).
d) Списки використовують фігурні дужки {}, а кортежі — квадратні [].
2. Яка структура даних у Python є прямою аналогією до об'єкта JSON ({...}) і слугує основою для роботи з API?
    a) Множина (set)
    b) Список (list)
    c) Словник (dict)
    d) Кортеж (tuple)
3. Вам потрібно отримати список унікальних слів (вокабуляр) з великого тексту. Яка структура даних найкраще та найефективніше впорається з цим завданням?
    a) Використати список і перевіряти наявність кожного слова перед додаванням.
    b) Перетворити список усіх слів на множину (set), яка автоматично видалить дублікати.
    c) Зберігати слова у кортежі.
d) Використати словник, де ключем і значенням буде одне й те саме слово.
4. Що таке рекурсія?
    a) Використання циклу for для обходу елементів.
    b) Процес, за якого функція викликає саму себе для вирішення меншої підзадачі.
    c) Спосіб імпортувати модулі в програму.
    d) Алгоритм сортування даних.
5. Який оператор використовується для перевірки, чи задовольняє змінна x одній з кількох умов (наприклад, x дорівнює 5 або x більше 10)?
    a) `if x == 5 and x > 10:`
    b) `if x == 5 & x > 10:`
    c) `if x == 5 or x > 10:`
    d) `if x in [5, 10]:`
6. Який є загальноприйнятим та рекомендованим способом імпортувати бібліотеку NumPy?
    a) `from numpy import *`
    b) `import numpy`
    c) `import numpy as np`
    d) `require numpy`
7. Який спосіб роботи з файлами в Python є найбільш надійним, оскільки він гарантує автоматичне закриття файлу навіть у разі помилки?
    a) `f = open('file.txt'); ...; f.close()`

b) with open('file.txt') as f: ...
c) f = open('file.txt', 'autoclose')
d) import files; files.open('file.txt')

8. При підрахунку частоти слів у тексті за допомогою словника, що зазвичай виступає в ролі ключа, а що — в ролі значення?
a) Ключ — це унікальне слово, значення — це кількість його повторень.
b) Ключ — це номер слова, значення — саме слово.
c) Ключ — це кількість повторень, значення — слово.
d) Ключ і значення — це одне й те саме слово.

9. Який тип даних матиме змінна x після виконання коду x = (42,)?
a) int (ціле число)
b) tuple (кортеж)
c) list (список)
d) set (множина)

10. Яка структура даних використовує ключі (а не числові індекси) для доступу до своїх елементів?
a) Список (list)
b) Кортеж (tuple)
c) Словник (dict)
d) Масив NumPy

Завдання для самостійної роботи

Це завдання допоможе вам закріпити на практиці роботу з основними структурами даних Python та зрозуміти їхню роль у вирішенні завдань, пов'язаних з обробкою даних.
Частина 1: Аналіз та рефлексія (Письмово)
Дайте розгорнуті відповіді на наступні питання.

1. Змінні vs Незмінні типи. У чому полягає фундаментальна різниця між списком (list) та кортежем (tuple)? Наведіть по одному прикладу, де використання списку є абсолютно необхідним, і де, навпаки, використання кортежу є більш доцільним та безпечним.
2. Вибір правильної структури. Ви працюєте над задачею аналізу тексту. Вам потрібно вирішити три підзадачі
   a) Зберегти всі слова з тексту, видаливши дублікати.
   b) Підрахувати, скільки разів кожне унікальне слово зустрічається в тексті.
   c) Зберегти послідовність речень у тому порядку, в якому вони йдуть у тексті.
   d) Яку з базових структур даних Python (list, tuple, dict, set) ви б обрали для кожної з цих підзадач і чому?
3. Рекурсія:
   Рекурсія є потужним інструментом для обходу деревних структур. Однак вона несе в собі ризик помилки "переповнення стека" (stack overflow). Поясніть своїми словами, що таке "базовий випадок" у рекурсивній функції і чому його наявність є критично важливою для уникнення нескінченної рекурсії.

Частина 2: Практичне завдання (Програма для аналізу тексту)
Напишіть скрипт на Python, який виконує наступні дії:

1. **Створює файл:** Програмно створіть текстовий файл з назвою my_article.txt і запишіть у нього невеликий текст (3-4 речення) на будь-яку тему. Текст повинен містити розділові знаки та слова в різних регістрах.
2. **Читає та обробляє файл:**
   - Відкрийте та прочитайте вміст файлу my_article.txt.
   - Приведіть весь текст до нижнього регістру.
   - Видаліть з тексту всі розділові знаки (наприклад, . , ! ?).
   - Розбийте текст на окремі слова.
3. **Аналізує дані:**
   - Створіть **словник**, щоб підрахувати частоту кожного слова в тексті.
   - Виведіть на екран 5 найпопулярніших слів та їхню частоту.
4. **Зберігає результат:**
   - Створіть **множину** всіх унікальних слів з тексту.
   - Запишіть ці унікальні слова (кожен з нового рядка) у новий файл з назвою vocabulary.txt.

Ключові навички для виконання: робота з файлами (with open), рядкові методи (.lower(), .replace(), .split()), робота зі словниками (dict) та множинами (set).

Частина 3: Дослідження та аналіз (Письмово)

Уявіть, що ви працюєте з даними від погодного API, які надходять у форматі JSON.

Приклад JSON-відповіді:

```
{
  "city": "Київ",
  "date": "2025-06-19",
  "temperature": 25.5,
  "weather": {
    "main": "Хмарно",
    "description": "Мінлива хмарність"
  },
  "forecast": [
    {"day": "П'ятниця", "temp": 26.0},
    {"day": "Субота", "temp": 27.5},
    {"day": "Неділя", "temp": 27.0}
  ]
}
```

Завдання:

1. Опишіть, як би ви, отримавши цей JSON у вигляді текстового рядка в Python, перетворили його на структури даних Python (словники та списки).
2. Напишіть короткі фрагменти коду (псевдокод або реальний код), щоб отримати з отриманої структури наступну інформацію:
   - Загальний опис погоди (значення ключа description).
   - Прогнозовану температуру на суботу.
   - Кількість днів у прогнозі.
3. Поясніть, чому саме комбінація словників та списків є настільки зручною для роботи з ієрархічними даними, як-от JSON.

Розділ 3. Масиви та тензори - мова спілкування з нейромережею

Вступ

За вражаючими досягненнями штучного інтелекту — від великих мовних моделей, що генерують текст, до нейронних мереж, які розпізнають зображення — стоять не лише складні алгоритми, а й фундаментальні структури даних. Це "скелет", на якому тримається весь інтелект системи.
Дані для ШІ — це не просто набір чисел чи слів. Щоб модель могла їх зрозуміти, виявити закономірності та зробити висновки, ці дані мають бути організовані у певну форму. Саме структури даних, такі як вектори, матриці, дерева та графи, дозволяють представити інформацію у вигляді, придатному для математичної обробки та аналізу. Вони перетворюють хаотичний потік даних на впорядковані системи, де можна простежити зв'язки, ієрархію та взаємовплив.
Нейронні мережі, попри всю їхню складність та інколи «магічну» поведінку, фундаментально є математичними моделями, що оперують виключно числами. Вони не розуміють, що таке "котик на фото" чи "позитивний відгук про товар". Щоб нейромережа могла обробити дані з реального світу, їх потрібно перекласти на мову, яку вона розуміє — мову чисел, організованих у чіткі структури. Цією мовою і є масиви та їх більш загальне уявлення — тензори. По суті, ці структури виступають універсальним перекладачем, що конвертує будь-які дані — від простого набору характеристик до кольорових зображень та відео — у впорядковані числові сітки, ідеально придатні для швидких паралельних обчислень на сучасних процесорах (GPU).
У цій темі ми розглянемо, як кожна з цих ключових структур даних слугує основою для різних галузей ШІ, від класичного машинного навчання до найсучасніших нейронних мереж, і чому правильний вибір структури є першим і найважливішим кроком на шляху до створення по-справжньому інтелектуальної системи.

3.1. Одновимірні масиви

Уявіть, що вам потрібно виконати просту математичну операцію — додати число 5 до кожного з мільйона елементів у списку.

```
# Підхід зі списками Python
python_list = list(range(1_000_000))

# Потрібно використовувати цикл
for i in range(len(python_list)):
    python_list[i] += 5
```

Хоча код простий, "під капотом" відбувається багато неефективної роботи. Інтерпретатор Python на кожній ітерації циклу змушений перевіряти тип даних, отримувати доступ до елемента, що може зберігатися будь-де в пам'яті, і виконувати операцію. Це повільно.

Рішенням є бібліотека NumPy та її основний об'єкт — багатовимірний масив (ndarray) , що зберігає елементи одного типу в неперервному блоці пам'яті. Ця однорідність та неперервність дозволяють застосовувати математичні операції („векторизацію“) над усім масивом одразу на низькому рівні, використовуючи оптимізований, компільований код (написаний на C або Fortran). Це в десятки та сотні разів швидше.

Властивості масиву:

- Фіксована довжина: кількість елементів визначається під час створення й не змінюється.
- Однорідність: усі елементи мають один тип (наприклад, цілі числа, дійсні, символи тощо).
- Безпосередній доступ: доступ до будь-якого елемента за індексом здійснюється за сталий час — *O(1)*.
- Послідовне розміщення: усі елементи зберігаються в послідовних комірках пам’яті.

***Таблиця 9**. Основні операції над масивами*

Операція	**Опис**	**Складність**
Доступ (access)	Отримання значення елемента за індексом	O(1)
Присвоєння (update)	Заміна значення елемента за індексом	O(1)
Пошук (search)	Лінійний пошук значення в масиві	O(n)
Вставка	Додати елемент у довільну позицію (вимагає зсув)	O(n)
Видалення	Видалити елемент зі зміщенням решти	O(n)
Ітерація	Обхід усіх елементів, наприклад, у циклі	O(n)

Приклад: Вектор ознак для опису квартири

- Ознаки: [площа (м²), кількість кімнат, відстань до метро (км), поверх]
- Вектор ознак для конкретного об'єкта: [75.0, 3.0, 1.2, 5.0]

Ефективність масиву кардинально залежить від операції, яку ви хочете виконати. У масиву є одна "суперсила" і одна "велика слабкість", які випливають з його фундаментальної властивості: масив — це неперервний блок пам'яті.

Сильна сторона, миттєвий доступ за індексом (O(1)) - це головна перевага масиву. Коли ви хочете отримати доступ до елемента за його індексом (наприклад, my_array[42]), комп'ютер не перебирає елементи з початку. Він виконує просту математичну операцію:

1. Бере початкову адресу масиву в пам'яті.
2. Знає розмір одного елемента (наприклад, 4 байти для цілого числа).
3. Множить індекс на розмір елемента і додає до початкової адреси.

Адреса_елемента = Початкова_адреса + (індекс * розмір_елемента)

Ця операція займає однаковий час, незалежно від того, чи звертаєтеся ви до 5-го елемента, чи до 5-мільйонного. Саме тому складність цієї операції позначається як O(1) — константний час. Аналогія: уявіть вулицю, де всі будинки пронумеровані і стоять щільно один за одним. Щоб знайти будинок №42, вам не потрібно заходити в кожен будинок з першого. Ви просто йдете прямо до 42-го.

Слабка сторона - повільна вставка та видалення (O(n)). Тут та сама властивість (неперервний блок пам'яті) стає недоліком.

- Вставка. Якщо ви захочете вставити новий елемент на початок або в середину масиву, ви не можете просто "втиснути" його туди. Щоб звільнити місце, комп'ютер змушений зсунути всі наступні елементи на одну позицію вправо. У найгіршому випадку (вставка на початок) доведеться зсунути всі n елементів.

- Видалення. Аналогічно, при видаленні елемента утворюється "дірка". Щоб її заповнити і зберегти неперервність, всі наступні елементи потрібно зсунути на одну позицію вліво.

Ці операції зсуву роблять вставку та видалення в середині масиву повільними, зі складністю O(n), оскільки час виконання прямо пропорційний кількості елементів, які потрібно зрушити.

***Таблиця 10**. Підсумкова таблиця ефективності*

Операція	**Часова складність**	**Пояснення**
Доступ за індексом	O(1)	Прямий доступ за адресою в пам'яті.
Пошук елемента	O(n)	У найгіршому випадку потрібно перевірити всі елементи (лінійний пошук).
Вставка / Видалення в кінці	O(1)*	Зазвичай це швидка операція (якщо є вільне місце).

Вставка / Видалення на початку / в середині	O(n)	Потребує зсуву великої кількості елементів.

Отже, масиви є надзвичайно ефективними, коли вам потрібен швидкий доступ до даних за відомим індексом і коли розмір колекції є відносно стабільним. Вони є поганим вибором для завдань, що вимагають частих вставок та видалень елементів у довільних місцях. Саме цей недолік і призвів до появи альтернативних структур даних, таких як зв'язні списки.

Одновимірний масив — це найпростіша структура, що зберігає послідовність елементів. У контексті ШІ він найчастіше використовується для представлення вектора ознак (feature vector) — числового "портрета" одного об'єкта.

Рис.15. Одновимірний масив та його властивості

Кожне число у векторі відповідає певній вимірюваній характеристиці (ознаці) цього об'єкта. Важливо, що навіть нечислові дані, такі як слова або категорії, для обробки моделлю також перетворюються на числові вектори за допомогою спеціальних технік (наприклад, one-hot encoding або word embeddings). Правильний вибір ознак є одним із найважливіших етапів у машинному навчанні, адже саме на їх основі модель шукатиме закономірності.

Приклад**.** Опис квартири для моделі прогнозування її вартості.

- Ознаки: площа (м²), кількість кімнат, відстань до метро (км), поверх.
- Об'єкт (квартира А): 75 м², 3 кімнати, 1.2 км до метро, 5-й поверх.
- Вектор ознак: [75.0, 3.0, 1.2, 5.0]

```
import numpy as np
```

```
# Створення вектора ознак для квартири за допомогою NumPy
apartment_features = np.array([75.0, 3.0, 1.2, 5.0])

print("Тип об'єкта:", type(apartment_features))
print("Розмірність:", apartment_features.ndim) # 1 вимір
print("Форма (кількість елементів):", apartment_features.shape) # (4,)
print("Вектор ознак:", apartment_features)
```

Саме такі вектори є вхідними даними для більшості класичних моделей машинного навчання та для повнозв'язних шарів нейронних мереж. Важливо зазначити, що в реальних задачах ці ознаки часто потребують нормалізації (приведення до спільного масштабу), щоб модель не надавала надмірної ваги ознакам з більшими числовими значеннями (наприклад, площі порівняно з кількістю кімнат).

Приклади опрацювання одновимірних масивів

1. Пошук елемента в масиві (лінійний пошук)

```
def search(array, target):
    for i, val in enumerate(array):
        if val == target:
            return i  # Знайдено індекс
    return -1  # Якщо не знайдено

a = [10, 25, 30, 45, 60]
print("Позиція:", search(a, 30))  # → 2
```

2. Вставка елемента в масив

```
def insert(array, index, value):
    return array[:index] + [value] + array[index:]

a = [1, 2, 3, 4]
a = insert(a, 2, 99)  # Вставляємо 99 на позицію 2
print(a)  # → [1, 2, 99, 3, 4]
```

3. Видалення елемента за індексом

```
def delete(array, index):
    return array[:index] + array[index+1:]

a = [1, 2, 3, 4, 5]
a = delete(a, 2)  # Видаляємо елемент з індексом 2 (число 3)
print(a)  # → [1, 2, 4, 5]
```

4. Перестановка елементів (swap)

```
def swap(array, i, j):
    array[i], array[j] = array[j], array[i]
    return array

a = [10, 20, 30, 40]
```

```
a = swap(a, 1, 3)  # Міняємо місцями елементи 20 і 40
print(a)  # → [10, 40, 30, 20]
```

5. Реверс масиву

```
def reverse(array):
    return array[::-1]

a = [1, 2, 3, 4, 5]
print(reverse(a))  # → [5, 4, 3, 2, 1]
```

Одновимірні масиви — це фундаментальна структура, особливо в контексті підготовки даних для алгоритмів машинного навчання. NumPy забезпечує ефективність і зручність, необхідну при роботі з великими обсягами числових даних. Вектор ознак — це базова одиниця представлення об'єкта у ШІ, і правильне її формування напряму впливає на якість моделі

3.2. Двовимірні масиви

Якщо одновимірний масив можна уявити як один ряд полиць, то двовимірний масив (2D array), або матриця, — це цілий стелаж, що має і рядки, і стовпці.

Це структура даних, яка організовує елементи у вигляді прямокутної сітки. Для доступу до будь-якого елемента тепер потрібно вказати два індекси: номер рядка та номер стовпця.

Структура та властивості двовимірного масиву:

- Фіксована форма (розмірність) - кожен підмасив (рядок) має однакову кількість елементів.
- Індексація - доступ до елемента виконується через пару індексів — arr[i][j], де i — індекс рядка, j — індекс стовпця.
- Послідовне зберігання - у низькорівневих мовах реалізується як один лінійний масив із розрахунком зміщення.
- Однорідність - усі елементи одного типу (наприклад, int, float).
- Підтримує вкладені ітерації — наприклад, подвійні цикли для обходу.

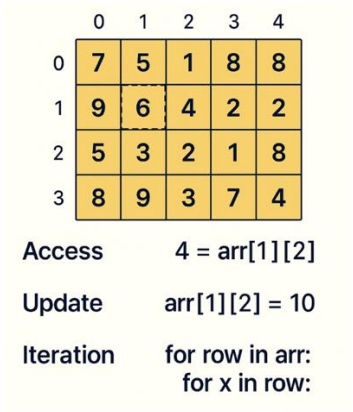


Рис.16. Двовимірний масив

Таблиця 11. *Набір основних операцій над двовимірними масивами*

Операція	**Приклад (Python)**	**Складність**
Ініціалізація	arr = [[0]*m for _ in range(n)]	O(n·m)
Доступ	x = arr[i][j]	O(1)
Присвоєння	arr[i][j] = 42	O(1)
Ітерація	подвійний цикл for i, for j	O(n·m)
Модифікація рядка	arr[i] = [1,2,3,4]	O(m)
Вставка рядка/стовпця	arr.insert(i, [...]) / вручну	O(n) або O(m)
Видалення	del arr[i][j] або pop	O(n) в гіршому
Пошук	Перебір у вкладеному циклі	O(n·m)

Одне з найпоширеніших застосувань двовимірнихмасивів у машинному наванні - представлення будь-яких наборів даних у вигляді таблиць (наприклад, з файлу Excel або CSV). Кожен рядок представляє один об'єкт або зразок (наприклад, одного пацієнта, одну квартиру, одного клієнта). Кожен стовпець представляє одну ознаку (feature) цього об'єкта (наприклад, вік, площа, ціна).

Приклад. Датасет пацієнтів
Маємо дані про трьох пацієнтів. Ознаки: [вік, зріст (см), вага (кг)].

- Пацієнт 1: 35 років, 180 см, 80.5 кг

- Пацієнт 2: 42 роки, 165 см, 65.0 кг
- Пацієнт 3: 28 років, 175 см, 72.3 кг

У вигляді матриці це виглядатиме так:

[

[35, 180, 80.5],
 [42, 165, 65.0],
 [28, 175, 72.3]

]

Така матриця має форму (3, 3), тобто 3 зразки та 3 ознаки.

Приклад**.** Набір даних з кількома квартирами.

Площа	**К-сть кімнат**	**Відстань до метро**	**Поверх**
75.0	3.0	1.2	5.0
45.5	1.0	0.3	9.0
110.0	4.0	2.5	3.0

```
# Набір даних у вигляді 2D масиву NumPy
dataset = np.array([
    [75.0, 3.0, 1.2, 5.0],
    [45.5, 1.0, 0.3, 9.0],
    [110.0, 4.0, 2.5, 3.0]
])

print("Розмірність масиву:", dataset.ndim) # 2 виміри
print("Форма (кількість зразків, кількість ознак):", dataset.shape) # (3, 4)
print("Набір даних:\n", dataset)
```

б) Представлення чорно-білих зображень

Чорно-біле (grayscale) зображення — це, по суті, сітка пікселів, де кожен піксель має певне значення яскравості (зазвичай від 0 — чорний, до 255 — білий). Таку сітку можна прямо представити у вигляді двовимірної матриці, зазвичай з типом даних uint8, що дозволяє ефективно зберігати цілочисельні значення яскравості в діапазоні від 0 до 255.

Приклад**.** Спрощене зображення цифри "1" розміром 5x5 пікселів.

```
# Представлення зображення у вигляді матриці, де 0 – чорний колір, а 255 – білий.
image_one = np.array([
    [0, 0, 255, 0, 0],
    [0, 0, 255, 0, 0],
```

```
    [0, 0, 255, 0, 0],
    [0, 0, 255, 0, 0],
    [0, 0, 255, 0, 0]
], dtype=np.uint8) # dtype для цілих чисел 0-255

print("Форма зображення:", image_one.shape) # (5, 5)
```

3.4. Опрацювання масивів як основа для попередньої обробки даних

У реальному світі дані рідко бувають ідеальними. Вони можуть містити помилки, пропуски, нерелевантну інформацію та шум. Якщо подати такі "сирі" дані безпосередньо в модель машинного навчання, її продуктивність буде низькою. Це ілюструє фундаментальний принцип: "Garbage In, Garbage Out" (Сміття на вході — сміття на виході).

Попередня обробка даних — це набір технік, що застосовуються для очищення, трансформації та підготовки даних, щоб зробити їх придатними для навчання моделі. Алгоритми опрацювання масивів є тим самим "хірургічним інструментарієм", за допомогою якого фахівець з даних перетворює хаос на порядок.

Розглянемо ключові операції на прикладі масиву NumPy.

Початкові дані:

```
import numpy as np

# Створимо матрицю 4x3, що представляє дані про 4-х студентів
# Стовпці: [Вік, Середній бал, Курс]
students_data = np.array([
    [20, 85, 3],
    [19, 92, 2],
    [21, 88, 4],
    [20, 78, 3]
])
print("Початковий масив даних:\n", students_data)
```

Вставка np.insert()

Вставка використовується для **інженерії ознак** (додавання нових стовпців) або для додавання нових зразків даних (рядків).

Важливо: np.insert повертає новий масив, не змінюючи оригінальний.

Приклад. Додавання рядка та стовпця

```
# Додавання нового студента (рядка)
new_student = np.array([18, 81, 1])
data_with_new_student = np.insert(students_data, 1, new_student, axis=0)
print("\n--- Додавання рядка ---")
print("Масив після додавання нового студента на позицію 1:\n",
data_with_new_student)

# Додавання нової ознаки (стовпця) - наприклад, бал за іспит
exam_scores = np.array([90, 95, 91, 85])
data_with_new_feature = np.insert(students_data, 2, exam_scores, axis=1)
print("\n--- Додавання стовпця ---")
print("Масив після додавання стовпця 'Бал за іспит' на позицію 2:\n",
data_with_new_feature)
```

Видалення np.delete()

Видалення дозволяє очищувати дані (видаляти некоректні рядки) та робити відбір ознак (видаляти неінформативні стовпці).

Важливо: np.delete також повертає новий масив.

Приклад. Видалення рядка та стовпця

```
# Видалення студента (рядка) з індексом 2, наприклад, через помилку в даних
cleaned_data_rows = np.delete(students_data, 2, axis=0)
print("\n--- Видалення рядка ---")
print("Масив після видалення студента з індексом 2:\n", cleaned_data_rows)

# Видалення ознаки "Вік" (стовпець з індексом 0) як нерелевантної
cleaned_data_cols = np.delete(students_data, 0, axis=1)
print("\n--- Видалення стовпця ---")
print("Масив після видалення стовпця 'Вік':\n", cleaned_data_cols)
```

Перестановка. Транспонування, обмін, перемішування

Перестановка змінює порядок даних, що є важливим для математичних операцій та для підготовки даних до навчання.

Приклад. Транспонування масиву (.T)
Транспонування міняє місцями рядки та стовпці. Це може бути корисним, наприклад, якщо ви хочете легко обчислити середнє значення для кожної ознаки, а не для кожного студента.

```
print("\n--- Транспонування ---")
transposed_data = students_data.T
print("Транспонований масив (ознаки тепер є рядками):\n", transposed_data)
```

Приклад. Обмін рядків/стовпців
Обмін може бути корисний для впорядкування даних за певними критеріями вручну.

```
# Створюємо копію, щоб не змінювати оригінал
data_to_swap = students_data.copy()
```

```
# Обмін рядків: міняємо місцями першого та останнього студентів
data_to_swap[[0, -1]] = data_to_swap[[-1, 0]]
print("\n--- Обмін рядків ---")
print("Масив після обміну першого та останнього рядків:\n", data_to_swap)

# Обмін стовпців: міняємо місцями "Вік" та "Курс"
data_to_swap[:, [0, 2]] = data_to_swap[:, [2, 0]]
print("\nМасив після обміну першого та останнього стовпців:\n", data_to_swap)
```

Приклад. Перемішування рядків (np.random.shuffle)

Це критично важливий крок перед навчанням більшості моделей ШІ. Він гарантує, що дані, які подаються в модель партіями (mini-batches), є репрезентативними і не відсортовані за якимось прихованим параметром.

```
# Створюємо копію
data_to_shuffle = students_data.copy()

# Перемішуємо рядки "на місці" (функція нічого не повертає, а змінює сам масив)
np.random.shuffle(data_to_shuffle)
print("\n--- Перемішування ---")
print("Масив після випадкового перемішування рядків:\n", data_to_shuffle)
```

Отже, базові операції з масивами — вставка, видалення та перестановка — є не просто технічними вправами, а ключовими інструментами в арсеналі фахівця з даних. Вони дозволяють перетворювати "сирі" дані на структурований, чистий та інформативний набір, готовий для подачі в модель машинного навчання, що безпосередньо впливає на якість кінцевого результату.

3.5. Від масиву до тензора: ключові концепції в ШІ

Між термінами масив, матриця та тензор існують важливі концептуальні відмінності. Розуміння цієї ієрархії є ключем до того, як "під капотом" працює сучасний штучний інтелект.

Двовимірний масив (2D Array): Це термін зі світу програмування. Він є структурою даних для зберігання елементів у вигляді сітки (рядки та стовпці) і слугує просто контейнером для даних. Його основне призначення — зберігання та організація інформації.

Матриця (Matrix): Це термін з математики, зокрема з лінійної алгебри. Як і масив, це прямокутна сітка чисел, але вона визначається набором математичних операцій (додавання, множення, транспонування). Основне призначення матриці —

представлення лінійних перетворень. Двовимірний масив є програмною реалізацією матриці.

Тензор (Tensor): Це найзагальніше поняття. У математиці та фізиці тензор — це узагальнення скалярів, векторів та матриць на будь-яку кількість вимірів.

- 0D тензор (ранг 0): Одне число, або скаляр.
- 1D тензор (ранг 1): Масив чисел, або вектор.
- 2D тензор (ранг 2): Масив векторів, або матриця.
- 3D тензор (ранг 3): Масив матриць, або "куб" даних.

У глибокому навчанні тензор — це основний об'єкт, з яким працюють фреймворки. Він є багатовимірним масивом, що має дві критично важливі для ШІ властивості, яких бракує масивам NumPy.

1. Апаратне прискорення (GPU/TPU Support). Тензори можуть бути легко переміщені на графічні (GPU) або тензорні (TPU) процесори для обчислень. Ці пристрої мають архітектуру, оптимізовану для паралельних обчислень, що дозволяє виконувати одну й ту саму операцію над тисячами елементів тензора одночасно. Це прискорює навчання моделей у сотні разів порівняно з CPU.
2. Автоматичне диференціювання (Autograd). Це найважливіша властивість. Тензори можуть "пам'ятати" операції, що над ними виконувалися, будуючи динамічний обчислювальний граф. Коли викликається метод .backward(), фреймворк проходить по цьому графу у зворотному напрямку, застосовуючи ланцюгове правило для автоматичного обчислення градієнтів (похідних). Цей процес, відомий як зворотне поширення помилки (backpropagation), є основою навчання нейронних мереж.

У програмуванні всі ці об'єкти представлені масивами. Але коли ми називаємо двовимірний масив матрицею, ми маємо на увазі, що будемо виконувати над ним математичні операції. А коли ми говоримо тензор (в контексті PyTorch/TensorFlow), ми маємо на увазі багатовимірний масив, який є фундаментальною одиницею для навчання нейронних мереж, здатною до обчислень на GPU та автоматичного диференціювання.

Уявіть нейронну мережу як обчислювальний граф, через який "течуть" тензори.

- Вхідні дані. Зображення, текст чи інші дані перетворюються на вхідний тензор.
- Параметри моделі. Ваги (W) та зміщення (b) кожного шару — це теж тензори. Саме їхні значення модель навчається підбирати під час тренування, і саме вони є навченою моделлю, що зберігається після навчання.
- Проміжні результати. Вихід кожного шару — це новий тензор, що слугує входом для наступного шару.
- Кінцевий прогноз. Результат роботи мережі — це вихідний тензор, що містить прогноз моделі.
- Функція втрат (Loss). Різниця між прогнозом та реальним значенням обчислюється і представляється у вигляді скалярного тензора (0D тензора). Зведення помилки до одного числа є критично важливим для алгоритмів оптимізації.
- Градієнти. За допомогою Autograd фреймворк обчислює тензори градієнтів для всіх параметрів моделі.

- Оновлення ваг. Оптимізатор використовує тензори градієнтів, щоб оновити параметри (ваги) моделі, роблячи крок у напрямку зменшення помилки.

Представлення складних даних за допомогою тензорів

а) 3D Тензор. Кольорове зображення

Чорно-біле зображення — це 2D матриця. Кольорове ж зображення зазвичай представляється у форматі RGB (Red, Green, Blue), де кожен піксель має три значення інтенсивності — для червоного, зеленого та синього каналів.

Отже, кольорове зображення — це, по суті, три двовимірні матриці, складені одна на одну. Це і є 3D тензор з формою `(висота, ширина, канали)`.

Приклад. Кольорове зображення 2x2 пікселі

```
import numpy as np
# Уявімо зображення 2x2, де кожен піксель має 3 значення (R, G, B)
# Наприклад: [червоний, зелений], [синій, білий]
color_image = np.array([
    [[255, 0, 0], [0, 255, 0]],    # Перший рядок пікселів
    [[0, 0, 255], [255, 255, 255]] # Другий рядок пікселів
])
print("3D Тензор (кольорове зображення):\n", color_image)
print("\nФорма тензора (висота, ширина, канали):", color_image.shape)
```

б) 4D Тензор. Партія (batch) зображень

Нейронні мережі майже ніколи не обробляють одне зображення за раз. Для ефективності та стабілізації навчання вони працюють з партіями (batches) з десятків чи сотень зображень.

Партія зображень — це масив 3D тензорів, що утворює 4D тензор з формою `(кількість_зображень, висота, ширина, канали)`.

в) 5D Тензор. Партія відео

Відео — це послідовність зображень (кадрів). Відповідно, одне відео — це 4D тензор `(кадри, висота, ширина, канали)`. А партія з кількох відео — це вже 5D тензор `(кількість_відео, кадри, висота, ширина, канали)`.

***Таблиця 12**. Порівняння двовимірних масивів, матриць та тензорів*

Характеристика	**Двовимірний масив**	**Матриця**	**Тензор (в ШІ)**
Галузь	Програмування	Математика	Штучний інтелект

Характеристика	Двовимірний масив	Матриця	Тензор (в ШІ)
Суть	"Структура даних, контейнер"	Математичний об'єкт з операціями	Багатовимірний масив з підтримкою GPU та Autograd
Розмірність	"Завжди 2 (рядки, стовпці)"	"Завжди 2 (рядки, стовпці)"	"Будь-яка (0D, 1D, 2D, 3D, ...)"
Основне завдання	Зберігання даних	Лінійні перетворення	"Потік даних у нейромережі, навчання"

Отже, тензор у ШІ — це багатовимірний масив, оптимізований для швидких паралельних обчислень та автоматичного знаходження градієнтів.

Контрольні запитання до розділу

1. Фундаментальна перевага NumPy. Поясніть своїми словами, чому операція numpy_array * 2 виконується набагато швидше, ніж аналогічна операція в циклі for для стандартного списку Python. Як поняття "неперервний блок пам'яті" та "векторизація" пов'язані з цією ефективністю?

2. Сутність вектора ознак. Уявіть, що вам потрібно описати музичний трек для рекомендаційної системи. Які характеристики ви б обрали (наприклад, тривалість, темп, наявність вокалу, жанр)? Як би виглядав вектор ознак для одного конкретного треку, і чому перетворення музики на такий числовий "портрет" є необхідним кроком для ШІ?

3. Багатовимірність даних. Опишіть, як би ви представили наступні дані у вигляді тензора, вказавши його ранг (кількість вимірів) та форму (shape):

- а) Партія з 64 чорно-білих зображень розміром 28x28 пікселів.
- б) Один аудіозапис тривалістю 10 секунд, де для кожної секунди є 16000 значень (амплітуда).

4. Від 2D до 1D і назад. Навіщо може знадобитися "розгортати" (flatten) двовимірний масив (матрицю), що представляє зображення, в одновимірний вектор? Яку інформацію про просторові зв'язки між пікселями ми при цьому втрачаємо?

5. Практика препроцесингу. Ви працюєте з табличним датасетом (представленим як 2D масив), де один зі стовпців — це "рік народження", а інший — "рік реєстрації". Яку нову ознаку ви могли б створити за допомогою операцій над цими стовпцями, і чому вона може бути більш корисною для моделі, ніж вихідні дані?

6. NumPy vs. Тензори в PyTorch/TensorFlow. Якщо масиви NumPy настільки ефективні, чому фреймворки глибокого навчання не використовують їх напряму, а мають власні об'єкти "тензори"? Назвіть дві ключові "суперсили" тензорів, яких немає у ndarray і які є критичними для навчання нейронних мереж.

7. Ефективність операцій. Функції np.insert та np.delete є зручними, але відносно "дорогими" (повільними) операціями. Чому вони працюють повільніше, ніж, наприклад, зміна значення існуючого елемента (arr[i, j] = value), і в якому випадку часте використання цих функцій може стати проблемою?

8. Пошук та логіка. У вас є матриця, що представляє результати медичного аналізу для 1000 пацієнтів, де один зі стовпців — рівень цукру в крові. Як за допомогою однієї операції (булевої індексації) ви можете отримати всі дані лише тих пацієнтів, у яких рівень цукру перевищує норму?

9. Зв'язок з архітектурою GPU. Чому жорстка, сіткоподібна структура масивів та тензорів ідеально підходить для паралельних обчислень на графічних процесорах (GPU)? Як це пов'язано з принципом SIMD (Single Instruction, Multiple Data)?

10. Рефлексія. Якби вам потрібно було пояснити різницю між вектором, матрицею та тензором людині, не знайомій з програмуванням, яку б аналогію з реального життя ви використали для кожного з цих понять?

Тести для самоконтролю

1. Чому масиви NumPy значно швидші за стандартні списки Python для числових обчислень?
    a) Тому що вони динамічно змінюють свій розмір.
    b) Тому що вони можуть зберігати дані різних типів.
    c) Тому що вони зберігають дані в неперервному блоці пам'яті та використовують векторизовані операції.
    d) Тому що вони мають простіший синтаксис.
2. Яка з цих властивостей є ключовою відмінністю тензорів у PyTorch/TensorFlow від масивів NumPy?
    a) Можливість зберігати лише цілі числа.
    b) Незмінність (immutability).
    c) Підтримка автоматичного диференціювання (autograd) та обчислень на GPU.
    d) Здатність бути лише одновимірними.
3. Як найприродніше представити чорно-біле зображення розміром 64x64 пікселів?
    a) Як 1D тензор (вектор) довжиною 4096.
    b) Як 2D тензор (матрицю) форми (64, 64).
    c) Як 3D тензор форми (64, 64, 1).
    d) Як скаляр.
4. Вектор ознак (feature vector) [вік, зріст, вага] для однієї людини є прикладом:
    a) 0D тензора

b) 1D тензора
c) 2D тензора
d) 3D тензора

5. Ви обробляєте партію (batch) з 32 кольорових зображень розміром 224x224 пікселів. Яку форму матиме тензор, що представляє ці дані?
a) (32, 224, 3)
b) (224, 224, 32, 3)
c) (32, 224, 224, 3)
d) (32, 3, 224)

6. Яку операцію потрібно виконати, щоб видалити стовпець з індексом 2 з NumPy масиву arr?
a) np.remove(arr, 2, axis=0)
b) np.delete(arr, 2, axis=1)
c) arr.pop(2)
d) np.delete(arr, 2, axis=0)

7. "Попередня обробка даних" (data preprocessing) — це:
a) Процес навчання моделі.
b) Процес оцінки якості моделі.
c) Етап підготовки та очищення "сирих" даних перед подачею в модель.
d) Процес візуалізації результатів.

8. Булева індексація в NumPy (наприклад, arr[arr > 5]) використовується для:
a) Перевірки, чи всі елементи є булевими.
b) Перетворення масиву на булевий тип.
c) Сортування масиву.
d) Фільтрації елементів масиву за певною умовою.

9. Якщо ви маєте NumPy масив A і виконуєте операцію B = A, а потім змінюєте елемент у B, що станеться з A?
a) A залишиться незмінним, оскільки B є копією.
b) A також зміниться, оскільки B є лише посиланням на той самий об'єкт у пам'яті.
c) Виникне помилка, оскільки масиви NumPy є незмінними.
d) B стане списком Python.

10. Транспонування матриці (наприклад, за допомогою arr.T) призводить до:
a) Зміни значень елементів на протилежні.
b) Перестановки елементів у випадковому порядку.
c) Видалення діагональних елементів.
d) Обміну місцями рядків та стовпців.

Завдання для самостійної роботи

Завдання спрямоване на закріплення практичних навичок роботи з багатовимірними масивами NumPy та розуміння їхньої ролі як основи для представлення даних у задачах ШІ.

Частина 1: Аналіз та рефлексія (Письмово)

Дайте розгорнуті відповіді на наступні питання.

1. NumPy чи Списки Python. Поясніть, у чому полягає ключова перевага масивів NumPy над стандартними списками Python при виконанні математичних операцій над великими наборами числових даних. Що таке "векторизація" і як вона пов'язана з ефективністю NumPy?
2. Тензори в глибокому навчанні. Масиви NumPy є чудовим інструментом для наукових обчислень. Чому тоді фреймворкам для глибокого навчання, як-от TensorFlow та PyTorch, знадобилося створювати власний об'єкт "тензор"? Назвіть та поясніть дві фундаментальні властивості тензорів, яких немає у масивів NumPy і які є критично важливими для навчання нейронних мереж.
3. Представлення даних. Опишіть, як би ви представили наступні типи даних у вигляді масивів/тензорів. Вкажіть необхідну кількість вимірів (ранг тензора) та форму (shape) для кожного випадку:
   - **а)** Вектор ознак, що описує один будинок (площа, кількість кімнат, рік побудови).
   - **б)** Чорно-біле зображення розміром 256x256 пікселів.
   - **в)** Партія (batch) з 16 кольорових (RGB) зображень розміром 128x128 пікселів.

Частина 2: Практичне завдання (Попередня обробка даних з NumPy)

Напишіть скрипт на Python з використанням бібліотеки NumPy, який симулює базовий конвеєр попередньої обробки даних.

1. Створення початкових даних:
   Створіть двовимірний NumPy масив розміром 5x4, який представляє дані про 5 продуктів. Стовпці: [ID_продукту, Ціна, Рейтинг, Кількість_на_складі]. Заповніть його довільними числовими даними.
2. Фільтрація та пошук:
   - Використовуючи булеву індексацію, виберіть та виведіть на екран тільки ті продукти, ціна яких вища за певне значення (наприклад, 1000).
   - Знайдіть та виведіть індекси продуктів, рейтинг яких дорівнює максимальному значенню в датасеті.
3. Вставка та видалення:
   - Уявіть, що ви вирішили додати нову ознаку — "Знижка" (у відсотках). Створіть новий стовпець (одновимірний масив) зі знижками для кожного продукту і вставте його як другий стовпець у вашу таблицю.
   - Видаліть стовпець ID_продукту, оскільки він не є корисною ознакою для навчання моделі.
4. Перестановка:
   - Перемішайте рядки у вашому масиві у випадковому порядку. Поясніть у коментарі до коду, чому перемішування даних є важливим етапом перед подачею їх у модель машинного навчання.

Виведіть результат на екран після кожного кроку, щоб продемонструвати трансформації масиву.

Частина 3: Дослідження та аналіз (Письмово)

Уявіть, що ви працюєте над моделлю комп'ютерного зору, яка повинна класифікувати зображення.

1. Опишіть, як кольорове зображення (наприклад, фотографія кота) перетворюється на тензор, зрозумілий для нейронної мережі. Які виміри матиме цей тензор і що кожен з них представляє?

2. Дослідіть та опишіть 2-3 типові операції попередньої обробки зображень (наприклад, нормалізація, зміна розміру, аугментація даних, як-от горизонтальне віддзеркалення).
3. Поясніть, які базові дії над елементами масивів (пошук, вставка, вилучення, перестановка, арифметичні операції) лежать в основі кожної з цих операцій обробки зображень.

Розділ 4. Від списків до графів: моделювання складних зв'язків

Вступ

Ми вже знаємо, як використовувати базові структури, такі як списки та словники, для зберігання та обробки даних. Вони чудово підходять для лінійних послідовностей або наборів пар "ключ-значення". Але що робити, коли зв'язки між даними набагато складніші?

Уявіть собі соціальну мережу, генеалогічне дерево, карту доріг або мережу веб-сайтів, пов'язаних посиланнями. У таких системах кожен елемент може бути пов'язаний з багатьма іншими, утворюючи складну павутину. Звичайні списки безсилі ефективно змоделювати таку структуру.

Саме тут на сцену виходять дерева та графи — структури даних, створені спеціально для представлення ієрархічних та мережевих зв'язків. У цій темі ми зробимо крок від простих лінійних колекцій до складних нелінійних систем. Ми дослідимо, як дерева дозволяють моделювати ієрархію, а графи — будь-які довільні зв'язки. Розуміння цих концепцій є ключовим для вирішення багатьох завдань у сфері логістики, біоінформатики, соціальних наук і, звісно, штучного інтелекту.

4.1. Зв'язні списки та їх класифікація

Коли ми говоримо про зберігання послідовностей даних, перше, що спадає на думку, — це масиви (або списки в Python). Вони прості та ефективні для доступу до елемента за індексом (операція O(1)). Уявіть масив як потяг, де вагони щільно зчеплені. Щоб додати новий вагон у середину, потрібно розчепити потяг і зсунути половину вагонів. Це означає, що операції вставки або видалення елемента в середині масиву є "дорогими" (операція O(n)), оскільки вимагають зсуву всіх наступних елементів.

Зв'язний список — це лінійна структура даних, яка вирішує цю проблему. Він схожий на ланцюжок "пошуку скарбів", де кожен скарб (вузол) має підказку, де шукати наступний. Замість зберігання елементів у суцільному блоці пам'яті, зв'язний список складається з окремих об'єктів, що називаються вузлами (nodes), які можуть бути розкидані по всій пам'яті. На відміну від масивів, зв'язні списки дозволяють легко додавати або видаляти елементи без необхідності зсувати інші дані.

Зв'язний список — це динамічна структура даних, що складається з елементів (вузлів), кожен з яких містить значення (дані, Data) і посилання (вказівник, Link/Pointer) на наступний елемент списку.

Особливості зв’язних списків

- Перший елемент називається головою списку.
- Останній елемент містить посилання, яке вказує на null (або None, в залежності від мови програмування), що означає кінець списку.

Така структура дозволяє вставляти та видаляти вузли, просто змінюючи посилання, без необхідності переміщувати інші елементи, що робить ці операції дуже швидкими (O(1)), якщо ми вже маємо посилання на потрібний вузол.

***Таблиця 13**. Порівняння продуктивності основних операцій у масивах та зв'язних списках (О-нотація)*

Операція	**Масив (Array)**	**Зв'язний список (Linked List)**
Доступ за індексом (Access)	O(1)	O(n)
Пошук елемента (Search)	O(n)	O(n)
Вставка/видалення на початку	O(n)	O(1)
Вставка/видалення в кінці	O(1)	O(1)

Важливо зазначити, що O(1) для вставки/видалення в кінці масиву є амортизованим часом (за умови, що не потрібна зміна розміру), а для зв'язного списку – за умови, що ми зберігаємо окреме посилання на останній елемент (хвіст).

Існує кілька основних типів зв'язних списків, кожен з яких має свої особливості та сценарії використання.

Однозв'язний список (Singly Linked List) - найпростіший тип зв'язного списку. Кожен вузол має одне посилання, яке вказує на наступний елемент. Останній вузол у списку вказує на None (або null), що сигналізує про кінець послідовності.

- Структура вузла: [Дані | Посилання на наступний]
- Рух: Тільки в одному напрямку — вперед, від початку (head) до кінця (tail).

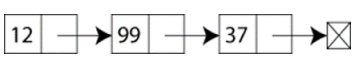


Рис.17. Однозв’язний список

Переваги:

- Простота реалізації та відлагодження.
- Мінімальні витрати пам'яті на вузол, оскільки зберігається лише одне додаткове посилання.

Недоліки:

- Неможливо рухатися у зворотному напрямку. Щоб дістатися до попереднього елемента, потрібно починати обхід з самого початку списку.
- Видалення елемента є неефективним (операція O(n)), якщо у вас є посилання тільки на сам вузол, що видаляється. Щоб його видалити, потрібно знайти попередній вузол, а для цього доведеться проходити весь список від початку.

Сценарій використання: Реалізація стека (Stack). Операції push (додавання в початок) та pop (видалення з початку) є дуже швидкими.

Двозв'язний список (Doubly Linked List) є більш гнучким. Кожен вузол має два посилання: одне на наступний елемент і одне на попередній.

- Структура вузла: [Посилання на попередній | Дані | Посилання на наступний]
- Рух: Можливий в обох напрямках — як вперед, так і назад.

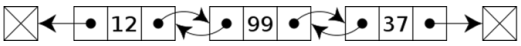


Рис.18. Двозв'язний список

Переваги:

- Дозволяє двонаправлений обхід, що значно спрощує деякі операції. Наприклад, при видаленні вузла B, ми можемо легко отримати доступ до його сусідів (A і C) і змінити їхні посилання, щоб вони вказували один на одного.
- Легко дістатися до попереднього елемента.

Недоліки**:**

- Кожен вузол вимагає більше пам'яті через додаткове посилання.
- Операції вставки та видалення трохи складніші, оскільки потрібно оновлювати два посилання замість одного, що збільшує кількість кроків і потенційних помилок.

Сценарій використання: Реалізація кешу з політикою LRU (Least Recently Used), де двозв'язний список дозволяє швидко переміщувати елементи, до яких нещодавно був доступ. Інший чудовий приклад — історія перегляду в браузері. Кожна веб-сторінка є вузлом. Посилання next дозволяє миттєво перейти на сторінку "вперед", а посилання prev — повернутися "назад". Це було б значно повільніше, якби історія зберігалася у звичайному масиві.

Кільцевий (циклічний) зв'язний список (Circular Linked List)

У кільцевому списку посилання останнього вузла вказує не на None, а на перший вузол (head), утворюючи таким чином замкнене кільце. Кільцевий список може бути як однозв'язним, так і двозв'язним.

- Структура: Подібна до одно- або двозв'язного, але останній елемент пов'язаний з першим.
- Рух: По колу, без кінця.

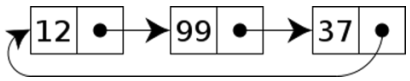


Рис.19. Кільцевий однозв'язний список

Переваги:

- Будь-який вузол може бути початковою точкою для обходу всього списку.
- Ефективно для додатків, де дані обробляються циклічно. Класичний приклад — планувальник процесів в операційній системі, який після виділення часу останньому процесу в черзі має миттєво перейти до першого, не перевіряючи кінець списку. Також використовується в іграх для передачі ходу між гравцями по колу.

Недоліки:

- Якщо не реалізувати обережно, обхід може призвести до нескінченного циклу. Для безпечного обходу потрібно зберегти посилання на початковий вузол (head) і продовжувати рух, доки поточний вузол знову не стане рівним початковому.
- Потрібно ретельно обробляти випадки, коли список порожній або містить один елемент, щоб коректно замкнути кільце.

***Таблиця 14**. Порівняння зв'язних списків*

Тип списку	**Структура посилань**	**Напрямок руху**	**Основні сценарії використання**
Однозв'язний	Одне (на наступний)	Тільки вперед	Реалізація стека, черги; прості випадки, де зворотний хід не потрібен.
Двозв'язний	Два (на наступний і попередній)	Вперед і назад	Реалізація кешу (LRU Cache), історія "вперед-назад" у браузері, текстові редактори (для операцій undo/redo).
Кільцевий	Останній вузол вказує на перший	По колу	Керування ресурсами по колу, циклічні плейлисти, ігри, де гравці ходять по черзі.

Вибір конкретного типу зв'язного списку залежить від вимог задачі: якщо потрібна максимальна простота і рух тільки вперед — підійде однозв'язний; якщо важлива гнучкість і двонаправлений обхід — двозв'язний; якщо дані мають циклічну природу — кільцевий.

4.2. Рівні абстракції: вказівники та посилання в Python

Програмісти, які переходять на Python з таких мов, як C або C++, часто ставлять питання: "А де тут вказівники?". Відповідь проста, але глибока: в Python немає вказівників у тому

вигляді, в якому вони існують у C++. Python свідомо абстрагується від прямого управління пам'яттю, щоб усунути цілий клас поширених помилок (таких як висячі вказівники або переповнення буфера), роблячи код значно безпечнішим та легшим для написання. Замість цього він надає більш високорівневу концепцію — посилання на об'єкти.

Розуміння цієї концепції є фундаментальним для написання коректного та ефективного коду на Python, оскільки вона пояснює, як насправді працюють змінні, присвоєння та передача аргументів у функції, і допомагає уникнути поширених помилок, пов'язаних з неочікуваною зміною даних.

Модель пам'яті Python: не «коробки зі значеннями», а «ярлики на об'єктах». У багатьох мовах програмування змінну можна уявити як "коробку" в пам'яті, в яку кладуть значення.
int x = 10; // У коробку x поклали число 10. Якщо ми потім напишемо int y = x;, буде створена нова коробка y, в яку скопіюється значення 10.

У Python модель інша. Усе в Python є об'єктом, а змінна — це лише ім'я (або "ярлик"), яке посилається на цей об'єкт у пам'яті. Цей підхід ідеально пасує динамічній природі мови.

x = 10 // Створюється об'єкт 10, і на нього вішається ярлик x.

Коли ви виконуєте y = x, ви не копіюєте значення з однієї коробки в іншу. Ви просто вішаєте другий ярлик (y) на той самий об'єкт. Обидва імені x та y тепер вказують на один і той самий об'єкт 10.

Для перевірки існує функція id(). Вбудована функція id() повертає унікальний ідентифікатор об'єкта, який зазвичай є його адресою в пам'яті. Це найближчий аналог вказівника, доступний у Python, який дозволяє перевірити, чи посилаються дві змінні на один і той самий об'єкт.

```
x = 100
y = x

print(f"Значення x: {x}, ID об'єкта, на який посилається x: {id(x)}")
print(f"Значення y: {y}, ID об'єкта, на який посилається y: {id(y)}")
# ID будуть однаковими!
```

Поведінка змінних кардинально відрізняється залежно від того, чи є об'єкт, на який вони посилаються, змінним (mutable) чи незмінним (immutable). До незмінних об’єктів (Immutable) належать: числа (int, float), рядки (str), кортежі (tuple), булеві значення (bool).

Коли ви намагаєтеся "змінити" незмінний об'єкт, насправді Python створює новий об'єкт у пам'яті, а змінна-ярлик просто перевішується на цей новий об'єкт.

```
x = 10
print(f"Початковий ID для x: {id(x)}")
```

```
x = x + 1 # Це не зміна об'єкта 10. Це створення нового об'єкта 11.
print(f"Новий ID для x: {id(x)}") # ID зміниться!
```

Старий об'єкт 10 залишається в пам'яті. Якщо на нього більше не посилається жоден ярлик, збирач сміття Python згодом його видалить. Цей механізм гарантує, що незмінні об'єкти є безпечними для використання в багатьох частинах програми одночасно — ви можете бути впевнені, що ніхто їх випадково не змінить.

До змінних рб’єктів (Mutable) належать: списки (list), словники (dict), множини (set). Коли ви змінюєте такий об'єкт, ви змінюєте вміст самого об'єкта, не створюючи нового. Усі ярлики, які посилалися на цей об'єкт, продовжать посилатися на нього ж, але тепер вони "бачитимуть" його змінений стан.

```
my_list_1 = [1, 2, 3]
my_list_2 = my_list_1 # Обидва ярлики посилаються на один і той самий список

print(f"Початковий ID для my_list_1: {id(my_list_1)}")
print(f"Початковий ID для my_list_2: {id(my_list_2)}") # ID однакові

# Змінюємо об'єкт через один з ярликів
my_list_2.append(4)

print(f"\nСписок my_list_1 після зміни: {my_list_1}") # [1, 2, 3, 4]
print(f"Список my_list_2 після зміни: {my_list_2}") # [1, 2, 3, 4]

print(f"\nКінцевий ID для my_list_1: {id(my_list_1)}") # ID не змінився!
print(f"Кінцевий ID для my_list_2: {id(my_list_2)}") # ID не змінився!
```

Це і є та поведінка, яка найбільше нагадує роботу з вказівниками, і саме її потрібно розуміти, щоб уникнути неочікуваних помилок.

Поширена пастка: змінні об'єкти як аргументи за замовчуванням.
Одним із класичних прикладів несподіваної поведінки є використання списку або словника як значення за замовчуванням для параметра функції. Такий об'єкт створюється лише один раз, під час визначення функції, і потім використовується в усіх її викликах.

```
def add_item(item, target_list=[]):
    target_list.append(item)
    return target_list

# Перший виклик працює, як очікується
print(add_item(1)) # Виведе: [1]

# Другий виклик додає елемент у той самий список!
print(add_item(2)) # Виведе: [1, 2]
```

Правильним підходом у такому випадку є використання None як значення за замовчуванням і створення нового списку всередині функції.

У Python використовується механізм передачі аргументів, відомий як «передача за присвоєнням» (pass-by-assignment). Це означає, що параметр функції стає новим «ярликом», який посилається на той самий об'єкт, що й аргумент.

- Якщо ви передаєте у функцію незмінний тип (число, рядок), то будь-яке "змінення" цього аргумента всередині функції створить новий локальний об'єкт. Оригінальний об'єкт поза функцією залишиться недоторканим.
- Якщо ви передаєте у функцію змінний тип (список, словник), то будь-яка модифікація самого об'єкта (наприклад, .append() для списку) всередині функції змінить той самий об'єкт, на який посилається змінна поза функцією.

```
def modify_data(some_number, some_list):
    some_number += 1
    some_list.append(100)

num = 5
data_list = [10, 20]
modify_data(num, data_list)

print(f"Після виклику функції: num={num}, data_list={data_list}")
# Результат: num=5, data_list=[10, 20, 100]
```

Розуміння моделі посилань є не просто академічною вправою, а критично важливою вимогою для ефективної роботи зі штучним інтелектом з кількох причин:

1. Ефективність при роботі з великими даними

Моделі ШІ навчаються на величезних наборах даних, які можуть займати гігабайти пам'яті. Це, як правило, об'єкти NumPy або Pandas, які є змінними. Коли ви передаєте такий датафрейм або масив у функцію для попередньої обробки, Python не копіює всі ці гігабайти. Він передає лише легке посилання на існуючий об'єкт.

- як наслідок, це робить обробку даних надзвичайно швидкою та ефективною з точки зору використання пам'яті. Якби Python копіював дані при кожному виклику функції, тренування моделей на великих датасетах було б практично неможливим.
- З іншого боку, є ризик, це "палиця з двома кінцями". Якщо функція ненавмисно змінює дані (наприклад, видаляє стовпець), ці зміни вплинуть на оригінальний об'єкт, що може призвести до важковідловлюваних помилок у подальших частинах коду. Тому часто використовують data.copy(), щоб свідомо створити копію для безпечних маніпуляцій.

2. Побудова конвеєрів обробки даних (Data Pipelines)

Робочий процес в ШІ часто виглядає як послідовність функцій: завантажити_дані() -> очистити_дані() -> додати_ознаки() -> навчити_модель().
Розуміння моделі посилань дозволяє чітко контролювати, чи кожна функція повертає новий, трансформований об'єкт, чи змінює існуючий "на місці" (in-place). Багато функцій у Pandas мають параметр inplace=True, який дозволяє свідомо обрати другий варіант для економії пам'яті.

3. Стан моделей у фреймворках глибокого навчання

Модель нейронної мережі в PyTorch або TensorFlow — це, по суті, великий змінний об'єкт. Її параметри (ваги та зміщення) зберігаються у тензорах, які також є змінними.

- Процес навчання — це ітеративна зміна цих тензорів-параметрів. Алгоритм зворотного поширення помилки обчислює градієнти, а оптимізатор (наприклад, Adam) використовує їх, щоб трохи змінити ваги. Ця зміна відбувається "на місці".
- Передача моделі - коли ви передаєте модель в оптимізатор (optimizer = Adam(model.parameters(), ...)), ви передаєте посилання на її навчальні параметри, дозволяючи оптимізатору їх змінювати на кожному кроці. Якби параметри передавалися за значенням, навчання було б неможливим.

Отже, модель , пам'яті Python базується на кількох ключових принципах:

1. Все є об'єктом - від чисел до функцій.
2. Змінні — це лише імена (ярлики), що посилаються на об'єкти.
3. Ключова відмінність лежить у типах об'єктів:
   - Незмінні (immutable) - операції над ними створюють нові об'єкти.
   - Змінні (mutable) - операції змінюють об'єкти «на місці».

   4. Передача аргументів — це передача посилань на об'єкти. Це означає, що змінні об'єкти можуть бути модифіковані всередині функції.

Для фахівця з ШІ це означає, що потрібно завжди пам'ятати про тип даних, з якими ви працюєте. Це дозволяє, з одного боку, використовувати переваги ефективної передачі великих об'єктів за посиланням, а з іншого — уникати непередбачуваних побічних ефектів, свідомо копіюючи дані там, де це необхідно для збереження їхньої цілісності.

4.3. Стеки і черги. Дек, як розширена версія черги

У програмуванні, окрім складних структур даних, таких як дерева та графи, існують простіші, але не менш важливі лінійні структури, що визначають порядок доступу до елементів. Серед них ключовими є стеки (stacks) та черги (queues). Вони є абстрактними типами даних (АТД). Це означає, що ми визначаємо їх через поведінку (тобто, які операції вони підтримують: push, pop, enqueue тощо), а не через конкретну реалізацію. Один і той самий АТД, як-от черга, може бути реалізований різними способами (наприклад, за допомогою масиву або зв'язного списку), і вибір реалізації впливає на його ефективність. Дек (deque), у свою чергу, є їх більш універсальним "родичем". Розуміння цих структур є фундаментальним, оскільки вони є "цеглинками" для побудови багатьох складніших алгоритмів.

1. Стек (Stack) — принцип LIFO

Стек — це структура даних, що працює за принципом LIFO (Last-In, First-Out), тобто "Останнім прийшов — першим пішов".

Аналогія: Уявіть стопку тарілок. Ви можете покласти нову тарілку тільки зверху, і взяти можете теж тільки верхню. Тарілка, яку ви поклали останньою, буде першою, яку ви візьмете.

Основні операції:

- push: Додати елемент на вершину стека.

- pop: Видалити та повернути елемент з вершини стека.
- peek (або top): Подивитися на елемент на вершині, не видаляючи його.
- is_empty: Перевірити, чи є стек порожнім.

Де використовується?

- Історія "Назад" у браузері: Кожна нова сторінка додається на вершину стека. Натискання кнопки "Назад" знімає верхній елемент.
- Функція скасування (Undo): Кожна дія користувача (наприклад, у текстовому редакторі) додається в стек. Операція "скасувати" просто виконує pop останньої дії.
- Стек викликів функцій: Це фундаментальний механізм роботи багатьох мов програмування. Коли функція A викликає функцію B, інформація про A (її локальні змінні та місце, куди повернутися) кладеться в стек. Потім у стек кладеться інформація про B. Коли B завершує роботу, вона видаляється зі стека, і виконання повертається до A.

Реалізація в Python:
Найпростіше реалізувати стек за допомогою звичайного списку (list), де append() виступає як push, а pop() без аргументів — як pop. Ці операції для списків є дуже ефективними (в середньому O(1)), оскільки вони працюють з кінцем масиву і не вимагають зсуву інших елементів. Важливо, що спроба реалізувати стек з іншого кінця списку (використовуючи list.insert(0) та list.pop(0)) була б такою ж неефективною, як і реалізація черги на списку.

```
# Імітація стека за допомогою списку
browser_history = []

# Користувач відвідує сторінки
browser_history.append('google.com')  # push
browser_history.append('wikipedia.org') # push
browser_history.append('python.org')    # push

print(f"Поточний стек: {browser_history}")

# Користувач натискає "Назад"
last_page = browser_history.pop() # pop
print(f"Повернулися зі сторінки: {last_page}")
print(f"Залишилось у стеку: {browser_history}")

# Дивимося, яка сторінка зараз на вершині
current_page = browser_history[-1] # peek
print(f"Поточна сторінка: {current_page}")
```

2. Черга (Queue) — принцип FIFO

Черга — це структура, що працює за принципом FIFO (First-In, First-Out), тобто "Першим прийшов — першим пішов".

Аналогія: Класична черга людей до каси в магазині. Той, хто прийшов першим, першим і буде обслужений.

Основні операції:

- enqueue: Додати елемент у кінець черги.
- dequeue: Видалити та повернути елемент з початку черги.
- front (або peek): Подивитися на перший елемент, не видаляючи його.
- is_empty: Перевірити, чи є черга порожньою.

Де використовується?

- Черга на друк: Документи стають у чергу і друкуються в тому порядку, в якому були відправлені.
- Обробка запитів на сервері: Запити від клієнтів обробляються в порядку їх надходження, щоб забезпечити справедливість.
- Алгоритми пошуку в ширину (BFS) для обходу дерев та графів, де потрібно досліджувати вузли рівень за рівнем.

Реалізація в Python:
Використовувати звичайний список для реалізації черги неефективно. Операція list.pop(0) (видалення з початку) вимагає зсуву всіх наступних елементів вліво на одну позицію, що має складність O(n). Правильним та ефективним інструментом для цього є об'єкт deque з модуля collections, який «під капотом» реалізований як двозв'язний список (doubly-linked list). У такій структурі кожен елемент зберігає посилання не тільки на наступний, але й на попередній елемент. Це дозволяє видаляти вузли з будь-якого кінця, просто змінюючи кілька посилань, без необхідності зсувати всі інші дані. Саме тому операції popleft() та append() для deque мають складність O(1).

```
from collections import deque

# Створюємо чергу
print_queue = deque()

# Додаємо завдання в чергу
print_queue.append('document1.pdf') # enqueue
print_queue.append('image.jpg')     # enqueue
print_queue.append('report.docx')   # enqueue

print(f"Поточна черга на друк: {print_queue}")

# Принтер бере перше завдання
first_in_queue = print_queue.popleft() # dequeue (ефективно)
print(f"Друкується: {first_in_queue}")
print(f"Залишилось у черзі: {print_queue}")

# Дивимося, хто наступний
next_doc = print_queue[0] # front
print(f"Наступний у черзі: {next_doc}")
```

3. Дек (Deque) — розширена версія черги

Дек (Deque, double-ended queue) — це узагальнення черги. Він дозволяє ефективно додавати та видаляти елементи з обох кінців.

Аналогія: Стопка карт, де ви можете швидко брати або класти карту як зверху, так і знизу.

Основні операції (у collections.deque):

- append(x): Додати x у правий кінець (як enqueue).
- appendleft(x): Додати x у лівий кінець.
- pop(): Видалити та повернути елемент з правого кінця.
- popleft(): Видалити та повернути елемент з лівого кінця (як dequeue).

Чому це важливо?
Дек є надзвичайно гнучкою структурою. За допомогою дека можна легко та ефективно реалізувати як стек, так і чергу:

- Стек: Використовуйте append() (як push) та pop() (як pop).
- Черга: Використовуйте append() (як enqueue) та popleft() (як dequeue).

Де використовується?

- Відстеження останніх елементів (sliding window): Дек ідеально підходить для реалізації «ковзного вікна» (sliding window) фіксованого розміру. Наприклад, при аналізі фінансових даних або потоку показників з датчиків, можна зберігати останні N вимірювань у деку. Коли надходить нове значення, воно додається праворуч (append), а найстаріше значення, що більше не потрібне для обчислення середнього, ефективно видаляється ліворуч (popleft).
- Алгоритми, що потребують доступу до обох кінців послідовності, наприклад, при пошуку паліндромів.

```
from collections import deque

# Створюємо дек
d = deque([3, 4, 5])

# Додаємо в лівий кінець
d.appendleft(2)
d.appendleft(1)
print(f"Дек після додавання ліворуч: {d}") # deque([1, 2, 3, 4, 5])

# Додаємо в правий кінець
d.append(6)
print(f"Дек після додавання праворуч: {d}") # deque([1, 2, 3, 4, 5, 6])

# Видаляємо з правого кінця
right_item = d.pop()
print(f"Видалено праворуч: {right_item}, залишилось: {d}")

# Видаляємо з лівого кінця
left_item = d.popleft()
print(f"Видалено ліворуч: {left_item}, залишилось: {d}")
```

Отже, вибір між цими структурами диктується «точкою доступу» до даних, яка потрібна вашому алгоритму. Потрібен доступ тільки до «вершини»? Використовуйте стек (LIFO).

Потрібно обробляти елементи в порядку прибуття? Використовуйте чергу (FIFO). Потрібен ефективний доступ до початку *і* кінця? Найкращим вибором буде дек. В Python для ефективної реалізації черг та деків варто завжди використовувати collections.deque, тоді як для простих стеків часто достатньо звичайного списку.

4.4. Хеш-таблиці: аналіз та застосування

У той час як лінійні структури даних, такі як масиви, забезпечують впорядкованість, вони програють в ефективності пошуку (O(n)). Хеш-таблиця пропонує інший підхід: жертвуючи впорядкованістю, вона забезпечує майже миттєвий доступ до даних (в середньому O(1)). Це робить її фундаментальною структурою для завдань, де швидкість доступу за ключем є пріоритетною.

Принцип функціонування

В основі архітектури хеш-таблиці лежать два ключові компоненти:

1. **Базовий масив (Array):** Структура, що складається з пронумерованих комірок, які часто називають "відрами" (buckets). Доступ до будь-якої комірки за її числовим індексом є операцією з константною складністю O(1).
2. **Хеш-функція (Hash Function):** Детермінований алгоритм, що перетворює вхідні дані довільного типу (ключ) на ціле число фіксованого розміру — **хеш-код**. Якість хеш-функції є критично важливою: вона повинна розподіляти ключі по комірках якомога більш рівномірно, щоб мінімізувати кількість колізій.

Наприклад:

"apple" → 12

"banana" → 7

"orange" → 3

Ти подаєш ключ (наприклад, "banana"), і хеш-таблиця майже миттєво каже тобі, що значення дорівнює 7.

h("banana") → 4

Це означає, що значення "banana" треба зберігати у позиції 4 в масиві.
Пошук, додавання і видалення даних займає в середньому O(1) часу — тобто майже миттєво. Але бувають випадки, коли різні ключі дають однаковий індекс — це називається колізія, і її треба правильно обробляти (наприклад, з допомогою зв’язних списків).

Процес доступу до даних включає наступні етапи:

1. Для заданого ключа обчислюється його хеш-код за допомогою хеш-функції.
2. Отриманий хеш-код трансформується в індекс комірки базового масиву. Зазвичай це досягається за допомогою операції взяття залишку від ділення на розмір масиву (hash_code % array_size).
3. Здійснюється прямий доступ до комірки масиву за обчисленим індексом.

Ключовою властивістю хеш-функції є її детермінізм: для однакових вхідних ключів вона завжди повинна генерувати ідентичні хеш-коди.

Колізія — це ситуація, коли хеш-функція генерує однаковий індекс для двох або більше різних ключів. Це є очікуваною поведінкою, і для її обробки існують різні стратегії. Найбільш поширеною є метод ланцюжків (chaining), за якого кожна комірка базового масиву є вказівником на іншу структуру даних, як правило, на зв'язний список. У разі колізії новий елемент (пара "ключ-значення") додається до цього списку. Таким чином, процес пошуку складається з двох кроків: обчислення індексу комірки та, за необхідності, лінійного пошуку в межах короткого зв'язного списку. При якісній хеш-функції та правильному коефіцієнті заповнення таблиці (load factor) довжина цих ланцюжків у середньому залишається дуже малою (близькою до 1). Хоча в найгіршому випадку, коли всі ключі потрапляють в одну комірку, складність операцій деградує до O(n), на практиці середня продуктивність залишається близькою до O(1)
У мові програмування Python хеш-таблиці є фундаментальною частиною і реалізовані у вигляді двох вбудованих типів даних:

- Словник (dict): Класична реалізація хеш-таблиці, що зберігає пари "ключ-значення".
- Множина (set): Спеціалізована реалізація, що зберігає лише унікальні ключі.

Виконання операції my_dict['key'] = value ініціює внутрішній механізм Python, який обчислює хеш-значення ключа 'key' для визначення відповідної комірки та збереження в ній значення value. Важливо, що ключем може бути лише незмінний (immutable) об'єкт (число, рядок, кортеж), оскільки його хеш-код не повинен змінюватися протягом життя об'єкта. Спроба використати змінний об'єкт, як-от список, як ключ (my_dict[[1, 2]] = 'value') призведе до помилки TypeError, оскільки вміст списку може змінитися, що зробило б його хеш нестабільним і порушило б логіку роботи таблиці.

Основне призначення хеш-таблиць полягає у забезпеченні швидкого доступу до даних за ключем. Це дозволяє уникнути неефективного лінійного пошуку в масивах або списках, особливо при роботі з великими обсягами даних.

```
# Повільний пошук (O(n)) у списку словників
users_list = [{'id': 101, 'name': 'Іван'}, {'id': 102, 'name': 'Марія'}]
# Для пошуку користувача з id=102 необхідно ітерувати по списку.

# Швидкий доступ (в середньому O(1)) у словнику
users_dict = {101: {'name': 'Іван'}, 102: {'name': 'Марія'}}
maria_data = users_dict.get(102) # Прямий доступ за ключем, .get() є
безпечнішим
```

Кешування (Caching) є технікою зберігання результатів обчислювально-інтенсивних операцій для їх повторного використання. Хеш-таблиці є ідеальним інструментом для цієї мети, де ключ представляє вхідні дані операції, а значення — її результат. Такий підхід називається мемоізацією.

- Приклад. Обчислення чисел Фібоначчі за допомогою рекурсії є класичною "дорогою" операцією. Мемоізація дозволяє уникнути повторних обчислень.

```
cache = {} # Хеш-таблиця для кешування
def fib(n):
```

```
if n in cache:
    return cache[n]
if n <= 1:
    return n
result = fib(n - 1) + fib(n - 2)
cache[n] = result # Зберігаємо результат у кеші
return result
```

Хеш-таблиці є незамінними в задачах обробки природної мови (Natural Language Processing).

- Побудова частотних словників. Для аналізу тексту часто створюється словник, де ключем є унікальне слово (токен), а значенням — кількість його входжень у текст.

```
text = "мова програмування Python це чудова мова"
words = text.split()
word_counts = {}

for word in words:
    word_counts[word] = word_counts.get(word, 0) + 1

# Результат: {'мова': 2, 'програмування': 1, 'Python': 1, 'це': 1, 'чудова': 1}
print(word_counts)
```

- Створення вокабулярів. На основі такого частотного словника або просто множини унікальних слів створюється вокабуляр, де кожному слову присвоюється унікальний числовий індекс (ID). Словник "слово -> ID", реалізований на хеш-таблиці, забезпечує миттєве перетворення слів на їхні індекси, що є обов'язковим першим кроком для подання тексту у числовому вигляді для моделі, як-от у техніках "мішок слів" (Bag of Words) або TF-IDF.

Таблиця 15. *Аналіз ефективності в задачах обробки природної мови*

Переваги	Недоліки
Висока швидкість: Середня часова складність для операцій вставки, видалення та пошуку становить O(1).	**Витрати пам'яті:** Для підтримки низького коефіцієнта заповнення (load factor) та зменшення кількості колізій хеш-таблиці можуть вимагати виділення більшого обсягу пам'яті, ніж це необхідно для безпосереднього зберігання даних.
Гнучкість ключів: Ключами можуть виступати будь-які хешовані (незмінні) типи	**Відсутність впорядкованості:** Класична хеш-таблиця не гарантує збереження порядку вставки елементів.*

даних, включаючи числа, рядки та кортежі.	
	Деградація продуктивності в найгіршому випадку: При виникненні великої кількості колізій (через погану хеш-функцію або специфічний набір даних) часова складність операцій може деградувати до O(n).

** Слід зазначити, що починаючи з версії Python 3.7, стандартна реалізація dict гарантує збереження порядку вставки. Проте ця властивість є специфічною для реалізації, а не фундаментальною характеристикою абстрактної структури даних "хеш-таблиця".*

Таким чином, хеш-таблиці є фундаментальною абстракцією, що лежить в основі багатьох сучасних систем. Вони є яскравим прикладом класичного інженерного компромісу, за якого свідома відмова від однієї властивості (впорядкованості) дозволяє отримати кардинальний виграш в іншій (швидкості доступу). Розуміння цього принципу є ключовим для розробки продуктивного та масштабованого програмного забезпечення.

4.5. Дерево рішень як один з найпростіших алгоритмів класифікації

Хоча лінійні структури, як-от списки, чудово підходять для зберігання послідовностей, вони неефективні, коли потрібно змоделювати ієрархію. Саме тут на сцену виходять дерева — один з найважливіших інструментів для представлення вкладених та ієрархічних зв'язків. Дерево — це ієрархічна структура даних, що складається з вузлів (вершин), з'єднаних ребрами (гілками). На відміну від графів, у деревах немає циклів, і кожен вузол (крім кореневого) має рівно одного "батька".

Ця структура ідеально підходить для представлення будь-яких ієрархічних зв'язків: файлової системи комп'ютера, структури компанії, генеалогічного дерева або, як ми розглянемо далі, логічного процесу прийняття рішень.

Дерево рішень (Decision Tree) — це модель машинного навчання, яка використовує деревну структуру для прийняття рішень. Це один з найпростіших та найбільш інтерпретованих алгоритмів. Його часто називають моделлю "білої скриньки" ("white box"), оскільки логіку його роботи можна легко візуалізувати та пояснити крок за кроком. Це є прямою протилежністю до "чорних скриньок", як-от складні нейронні мережі, де навіть розробникам буває складно зрозуміти, чому модель прийняла те чи інше рішення.

Аналогія: Уявіть, що ви вирішуєте, чи брати з собою парасольку. Ваша логіка може виглядати так:

1. Подивитися у вікно: "На небі є хмари?"
    - Так: "Йде дощ?"

        - Так: Взяти парасольку.
        - Ні: Перевірити прогноз погоди. "Ймовірність дощу > 30%?"
            - Так: Взяти парасольку про всяк випадок.
            - Ні: Не брати парасольку.
    - Ні: Не брати парасольку.

Це і є просте дерево рішень, що розбиває складну задачу на послідовність простих питань.

Структура Дерева рішень

- Кореневий вузол (Root Node): Найперший та найважливіший вузол, що представляє головну ознаку, за якою починається розділення даних.
- Внутрішні вузли (Internal Nodes): Вузли, що представляють певну ознаку (питання). Кожен такий вузол далі розбиває дані на підгрупи.
- Гілки (Branches): Представляють результат перевірки умови (відповідь на питання, наприклад, "так/ні" або ">5 / <=5"). Кожна гілка веде до наступного вузла.
- Листя (Leaf Nodes): Кінцеві вузли, що не мають нащадків. Вони містять кінцеве рішення — клас (у задачах класифікації) або середнє числове значення (у задачах регресії).

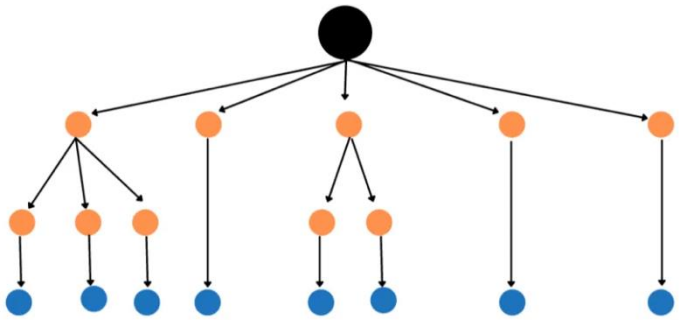


Рис. 20. Дерево рішень

Як працює дерево рішень (на прикладі класифікації)? Мета алгоритму — побудувати таке дерево, яке б найкращим чином розділяло дані на "чисті" групи, де в кожній групі були б представники переважно одного класу. Алгоритм побудови дерева є "жадібним" (greedy). Це означає, що на кожному кроці він обирає найкраще можливе розділення на даний момент, не заглядаючи наперед, як цей вибір вплине на наступні кроки. Такий підхід є швидким та ефективним, але не гарантує знаходження глобально оптимального дерева.

Процес навчання (побудови дерева):

1. Вибір найкращої ознаки: Алгоритм аналізує всі наявні ознаки (наприклад, "колір", "розмір", "форма" для фруктів) і обирає ту, яка найкраще розділяє дані на групи. Для оцінки "якості" розділення використовуються такі критерії:
    - Індекс Джині (Gini Impurity): Вимірює ймовірність того, що випадково обраний елемент з групи буде неправильно класифікований. Значення 0 означає ідеальну чистоту (всі елементи одного класу), 0.5 — максимальну невизначеність. Алгоритм прагне мінімізувати цей індекс.

- Інформаційний приріст (Information Gain): Базується на понятті ентропії з теорії інформації. Ентропія вимірює рівень "хаосу" або "невизначеності" в групі. Алгоритм обирає ознаку, яка дає максимальне зменшення ентропії (тобто максимальний приріст інформації) після розділення. На практиці обидва критерії зазвичай дають дуже схожі результати, хоча індекс Джині обчислюється трохи швидше, оскільки не містить логарифмів.

2. Розділення даних: На основі обраної ознаки та її значень дані розділяються на підгрупи.
3. Рекурсія: Процес повторюється для кожної підгрупи — знову шукається найкраща ознака для подальшого розділення, створюються нові вузли та гілки.
4. Зупинка: Рекурсія зупиняється, коли виконується одна з умов:

- Усі елементи в групі належать до одного класу (група є "чистою", індекс Джині = 0).
- Досягнуто максимальної глибини дерева (гіперпараметр max_depth, що задається для уникнення перенавчання).
- У групі залишилося занадто мало елементів для подальшого розділення (гіперпараметр min_samples_leaf).
- Подальше розділення не дає суттєвого приросту інформації.

Приклад. Класифікація фруктів

Уявімо, що нам потрібно класифікувати фрукти як "Яблуко" або "Лимон" на основі їхнього кольору та діаметра.

Дані:

- Фрукт 1: Червоний, діаметр 8 см -> Яблуко
- Фрукт 2: Жовтий, діаметр 6 см -> Лимон
- Фрукт 3: Зелений, діаметр 7 см -> Яблуко
- Фрукт 4: Жовтий, діаметр 8 см -> Яблуко
- Фрукт 5: Жовтий, діаметр 5.5 см -> Лимон

Побудоване дерево може виглядати так:

1. (Кореневий вузол) Питання: "Діаметр > 7.5 см?"
   - Так (Гілка 1): -> (Лист) Прогноз: Яблуко. (Сюди потрапили фрукти 1 і 4. Група "чиста").
   - Ні (Гілка 2): -> (Внутрішній вузол) Питання: "Колір жовтий?" (Сюди потрапили фрукти 2, 3, 5. Група "нечиста", тому потрібне подальше розділення).
     - Так (Гілка 2.1): -> (Лист) Прогноз: Лимон. (Сюди потрапили фрукти 2 і 5. Група "чиста").
     - Ні (Гілка 2.2): -> (Лист) Прогноз: Яблуко. (Сюди потрапив фрукт 3. Група "чиста").

Тепер, якщо ми отримаємо новий фрукт (наприклад, жовтий, діаметр 7 см), ми можемо легко його класифікувати, пройшовшись по дереву:

1. Діаметр > 7.5 см? Ні. -> Ідемо по гілці 2.
2. Колір жовтий? Так. -> Ідемо по гілці 2.1.

Прогноз: Лимон.

***Таблиця 16**. Переваги та недоліки дерева рішень*

Переваги	**Недоліки**
Висока інтерпретованість: Легко зрозуміти та візуалізувати логіку прийняття рішень. Це дозволяє не тільки робити прогнози, а й отримувати знання про дані.	Схильність до перенавчання (Overfitting): Якщо не обмежувати глибину, дерево може стати надто складним і "вивчити напам'ять" навчальні дані, включно з їхніми шумами та випадковостями. В результаті воно буде чудово працювати на даних, які вже бачило, але погано узагальнюватиме закономірності на нових, невідомих даних.
Невимогливість до підготовки даних: Не потребує масштабування чи нормалізації ознак, оскільки працює з пороговими значеннями.	Нестабільність: Невеликі зміни у вхідних даних (наприклад, додавання кількох нових записів) можуть призвести до побудови зовсім іншого дерева. Ця властивість називається високою дисперсією (high variance).
Універсальність: Може працювати як з числовими, так і з категоріальними даними без необхідності їх попереднього перетворення.	Може створювати зміщені (biased) дерева, якщо деякі класи домінують у даних. Також алгоритм схильний надавати перевагу ознакам з великою кількістю унікальних значень.

Таким чином, дерево рішень є втіленням компромісу між простотою та точністю. Хоча окреме дерево може поступатися в точності більш складним моделям, його прозорість є незамінною для аналізу даних. Водночас його головна сила розкривається саме в тому, що воно є будівельним блоком для більш складних та точних ансамблевих методів. Такі техніки, як випадковий ліс (Random Forest), який усереднює результати багатьох різних дерев, та градієнтний бустинг (Gradient Boosting), який послідовно виправляє помилки попередніх дерев, використовують слабкості окремих дерев (їхню нестабільність) як перевагу, створюючи надзвичайно точні та надійні моделі.

4.6. Піраміда як структура даних та основа для пірамідального сортування

У світі структур даних піраміда (Heap) займає особливе місце. Це спеціалізована деревоподібна структура даних, яка задовольняє певним властивостям і є надзвичайно ефективною для задач, пов'язаних із пошуком та вилученням максимального або мінімального елемента. Саме ця властивість робить піраміду не тільки основою для ефективного алгоритму сортування (Heapsort), але й ключовим компонентом для реалізації черг з пріоритетом, що використовуються в багатьох алгоритмах оптимізації.

Піраміда (Heap)— це структура даних, яку можна візуалізувати як бінарне дерево, що задовольняє двом ключовим умовам:

1. Умова форми (Shape Property) - це майже повне бінарне дерево. Це означає, що всі рівні дерева повністю заповнені, за винятком, можливо, останнього, який заповнюється зліва направо без пропусків. Ця сувора вимога до форми є критично важливою, оскільки саме вона дозволяє дуже ефективно представляти піраміду у вигляді звичайного масиву, уникаючи потреби у складних об'єктах вузлів з посиланнями.
2. Умова піраміди (Heap Property) - значення в кожному вузлі задовольняє певному відношенню зі значеннями його нащадків. Існує два типи пірамід:
   - Max-Heap (Макс-піраміда) - значення кожного батьківського вузла більше або дорівнює значенням його нащадків. Це гарантує, що найбільший елемент завжди знаходиться у корені дерева (на першій позиції в масиві).
   - Min-Heap (Мін-піраміда) - значення кожного батьківського вузла менше або дорівнює значенням його нащадків. Це гарантує, що найменший елемент завжди знаходиться у корені.

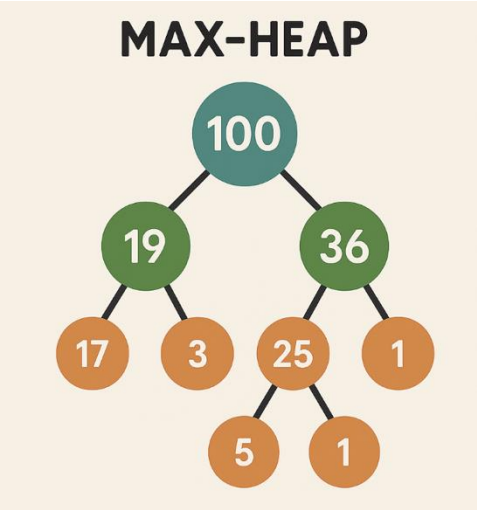


Рис. 21.Піраміда Max-Heap

Завдяки умові форми, піраміду дуже зручно зберігати у звичайному масиві, не використовуючи складні об'єкти вузлів з вказівниками. Для будь-якого елемента в масиві з індексом i:

- Його батько знаходиться за індексом (i - 1) // 2.
- Його лівий нащадок знаходиться за індексом 2 * i + 1.
- Його правий нащадок знаходиться за індексом 2 * i + 2.

Це дозволяє швидко переміщатися по дереву, виконуючи прості арифметичні операції з індексами, що є значно ефективнішим, ніж слідування за посиланнями в пам'яті.

Пірамідальне сортування (Heapsort)

Heapsort — це ефективний алгоритм сортування "на місці" (in-place), який використовує структуру Max-Heap для впорядкування елементів. Процес складається з двох основних фаз.

Для прикладу візьмемо масив: [4, 10, 3, 5, 1]

Фаза 1: Побудова Max-Heap (Heapify)

Перший крок — перетворити початковий невпорядкований масив на Max-Heap. Цей процес, який також називають heapify, полягає у послідовному застосуванні операції 'просіювання вниз' (sift down) до всіх нелистових вузлів, починаючи з останнього і рухаючись до кореня. Операція heapify для вузла гарантує, що він та всі його нащадки задовольняють умові піраміди.

Після побудови Max-Heap наш масив буде виглядати так: [10, 5, 3, 4, 1]. Тепер ми гарантовано знаємо, що найбільший елемент (10) знаходиться на першій позиції (у корені).

Фаза 2: Сортування

Друга фаза полягає в тому, щоб послідовно вилучати найбільший елемент з піраміди і ставити його на правильне місце в кінці масиву. Цей процес можна уявити як поступове "зменшення" піраміди та "зростання" відсортованої частини масиву.

1. Крок 1:
   - Міняємо місцями кореневий елемент (10) з останнім елементом у піраміді (1). Масив стає: [1, 5, 3, 4, 10].
   - Елемент 10 тепер на своєму фінальному, відсортованому місці. Ми "зменшуємо" розмір піраміди на один елемент, умовно "відрізаючи" 10 від неї.
   - Структура Max-Heap порушена, оскільки в корені тепер стоїть 1. Застосовуємо операцію 'просіювання вниз' (heapify) до кореня, щоб новий елемент 'опустився' на свою правильну позицію, відновлюючи властивість піраміди для зменшеного масиву (тепер розміром 4). Масив стає: [5, 4, 3, 1, 10].
2. Крок 2:
   - Найбільший елемент серед решти — 5. Міняємо його з останнім елементом піраміди (1). Масив стає: [1, 4, 3, 5, 10].
   - Елемент 5 тепер на своєму місці. Розмір піраміди знову зменшується.
   - Застосовуємо heapify до кореня. Масив стає: [4, 1, 3, 5, 10].
3. Крок 3:
   - Найбільший елемент — 4. Міняємо його з 3. Масив стає: [3, 1, 4, 5, 10].
   - Розмір піраміди зменшується.
   - Відновлюємо піраміду: [3, 1, 4, 5, 10] (оскільки 3 більше за 1, структура не змінюється).
4. І так далі... Процес повторюється, поки всі елементи не будуть вилучені з піраміди і не займуть свої відсортовані позиції.

Кінцевий результат буде: [1, 3, 4, 5, 10].

***Таблиця 17**. Переваги та недоліки пірамідального сортування*

Переваги	Недоліки
Висока ефективність: Часова складність завжди становить O(n log n), як у найкращому, так і в	Нестабільність: Алгоритм не є стабільним, тобто він не гарантує збереження початкового порядку для елементів з однаковими

Переваги	Недоліки
середньому та найгіршому випадках. Це робить Heapsort дуже надійним вибором, коли потрібно гарантувати продуктивність.	значеннями (наприклад, якщо у вас є два об'єкти з однаковим пріоритетом, їхній відносний порядок після сортування може змінитися). Це може бути проблемою в деяких задачах, де початковий порядок важливий.
Ефективність по пам'яті: Це алгоритм сортування "на місці" (in-place). Йому не потрібна додаткова пам'ять для зберігання відсортованих даних, оскільки він використовує сам вхідний масив. Простір пам'яті — O(1). Це робить його ідеальним для роботи з великими масивами в умовах обмеженої пам'яті.	Погане кешування: Доступ до елементів відбувається не послідовно, а стрибками по масиву (наприклад, від батька до далеких нащадків), що не дуже добре для сучасних процесорних кешів, які оптимізовані для роботи з сусідніми даними. Це може робити Heapsort повільнішим на практиці, ніж, наприклад, Merge Sort, попри однакову асимптотичну складність.

Варто зазначити, що піраміда є найпоширенішою та найефективнішою реалізацією абстрактної структури даних 'черга з пріоритетом' (Priority Queue). Саме в цій ролі вона стає ключовим компонентом багатьох важливих алгоритмів. Наприклад, в алгоритмі Дейкстри для пошуку найкоротшого шляху в графі черга з пріоритетом (реалізована як Min-Heap) дозволяє на кожному кроці миттєво обирати найближчий ще не відвіданий вузол. В операційних системах вона використовується для планування процесів, надаючи пріоритет більш важливим завданням. Розуміння принципів роботи піраміди відкриває двері до вирішення цілого класу оптимізаційних задач.

4.7. Графи. Нейронна мережа — це граф

У сучасному світі, де дані все частіше представляють собою складні мережі — від соціальних зв'язків до молекулярних структур — виникає потреба у структурах даних, здатних ефективно моделювати ці неієрархічні відношення. Якщо дерево чудово представляє ієрархію, то граф є універсальною мовою для опису будь-яких мереж.

Ця концепція є настільки фундаментальною, що лежить в основі найпотужніших інструментів сучасного штучного інтелекту. І головне, що потрібно зрозуміти: нейронна мережа — це і є граф.

Граф — це математична структура, що складається з двох основних елементів:

1. Вершини (або вузли, nodes): Представляють об'єкти або сутності.
2. Ребра (або зв'язки, edges): Представляють зв'язки або відношення між цими об'єктами.

Аналогії з реального світу:

- Соціальна мережа: Кожна людина — це вершина, а дружба між людьми — це ребро.
- Карта доріг: Кожне місто — це вершина, а дорога між містами — це ребро.
- Інтернет: Кожен веб-сайт — це вершина, а гіперпосилання з одного сайту на інший — це ребро.

Основні типи графів:

- Неспрямований (Undirected), коли ребра не мають напрямку (дружба в Facebook є взаємною).
- Спрямований (Directed), коли ребра мають напрямок (підписка в Instagram або Twitter не обов'язково є взаємною).
- Зважений (Weighted), коли кожне ребро має "вагу" або "ціну" (наприклад, відстань між містами на карті).

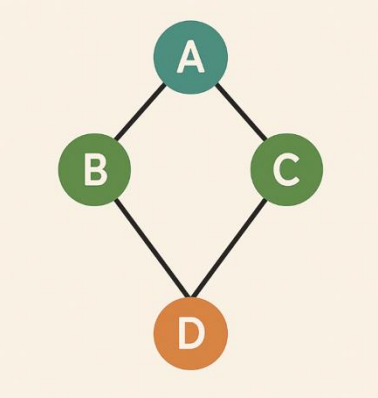


Рис.22. Простий граф з вершинами A, B, C, D та ребрами, що їх з'єднують

Нейронна мережа — це обчислювальна модель, натхненна роботою біологічного мозку. Вона складається з шарів штучних нейронів, з'єднаних між собою.

- Нейрони (Neurons): Прості обчислювальні одиниці, що отримують сигнали, обробляють їх і передають далі.
- Зв'язки (Synapses): З'єднання між нейронами, через які передаються сигнали. Кожен зв'язок має вагу (weight), яка визначає силу цього сигналу.

Інформація проходить через мережу від вхідного шару, через один або кілька прихованих шарів, до вихідного шару, який дає кінцевий результат (наприклад, класифікацію зображення).

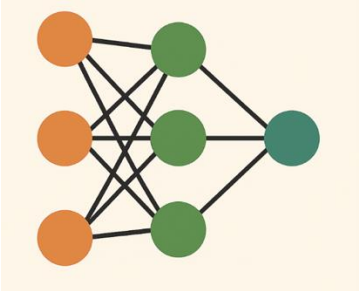


Рис.23. Проста нейронна мережа з вхідним, прихованим та вихідним шарами

Поєднаємо ці два поняття. Якщо ми подивимося на структуру нейронної мережі, ми побачимо, що вона ідеально описується мовою графів.

- Нейрони — це вершини (вузли) графа.
- Зв'язки між нейронами — це ребра графа.

Більше того, нейронна мережа є специфічним типом графа:

1. Це спрямований граф. У стандартних мережах прямого поширення (feedforward networks) інформація рухається суворо в одному напрямку, утворюючи спрямований ациклічний граф (DAG). У рекурентних нейронних мережах (RNN) існують цикли, що дозволяють моделі зберігати інформацію про попередні стани, але навіть їх можна „розгорнути в часі“ у вигляді DAG для аналізу.
2. Це зважений граф. Кожне ребро (зв'язок) має вагу. Ця вага — не просто число, а ключовий параметр моделі. Вона визначає, наскільки важливим є сигнал від одного нейрона для іншого. Сигнал, що проходить через ребро, множиться на його вагу. Якщо вага велика, сигнал посилюється; якщо мала або від'ємна — послаблюється або інгібується.
3. Це обчислювальний граф (Computational Graph). У ширшому сенсі, не лише нейрони, а й самі математичні операції (множення, додавання, функції активації) є вершинами в обчислювальному графі. Нейрони є лише композицією цих базових операцій. Тензори даних „течуть“ по ребрах від однієї операції до іншої. Ця функція вводить у модель нелінійність, що дозволяє їй вивчати складні закономірності, які неможливо описати простими лінійними рівняннями.

Процес навчання нейронної мережі — це процес налаштування графа.
Коли ми навчаємо нейронну мережу, її структура (кількість вершин та ребер) зазвичай залишається незмінною. Навчання полягає в тому, щоб ітеративно коригувати ваги на ребрах цього графа. Алгоритм зворотного поширення помилки (backpropagation) обчислює, наскільки кожна вага вплинула на кінцеву помилку, і трохи змінює її, щоб наступного разу помилка була меншою. Таким чином, весь процес навчання можна розглядати як задачу оптимізації, де ми шукаємо такий набір ваг для ребер графа, який мінімізує помилку на наших даних.
Таким чином, навчена нейронна мережа — це зважений спрямований граф, налаштований на вирішення конкретної задачі.

Чому це розуміння є важливим для ШІ? Розгляд нейронної мережі як графа дозволяє зрозуміти її математичну суть. Це не "чорна скринька", а обчислювальна структура, де дані "течуть" по ребрах, а обчислення відбуваються у вершинах. Це допомагає візуалізувати потік інформації та зрозуміти, як різні шари взаємодіють між собою. Бібліотеки, як-от TensorFlow та PyTorch, „під капотом“ будують саме обчислювальні графи. Коли ви визначаєте архітектуру мережі, ви, по суті, описуєте структуру графа. Саме ця графова структура дозволяє застосовувати алгоритм зворотного поширення помилки: фреймворк проходить по графу у зворотному напрямку, використовуючи ланцюгове правило диференціювання для автоматичного обчислення градієнтів. Крім того, це дозволяє оптимізувати обчислення для паралельного виконання на GPU.

Якщо сама нейронна мережа — це граф, то що робити, якщо вхідні дані також є графом (наприклад, соціальна мережа, молекула, 3D-модель)? Для цього були створені графові нейронні мережі. Це спеціальний тип ШІ, який вміє працювати безпосередньо з даними, представленими у вигляді графів, аналізуючи не лише властивості об'єктів (вузлів), але і структуру їхніх зв'язків. GNNs є потужним інструментом для задач, де контекст та

зв'язки між даними є ключовими, наприклад, у рекомендаційних системах або при розробці ліків.

Отже, нейронна мережа — це не просто метафора графа. Це і є граф: зважений, спрямований, обчислювальний граф, де процес навчання — це ітеративна оптимізація ваг на його ребрах. Цей погляд дозволяє не тільки демістифікувати глибоке навчання, а й відкриває шлях до створення нових архітектур ШІ, здатних працювати з даними, що мають складну реляційну структуру, наближаючи нас до створення більш потужних та контекстуально обізнаних інтелектуальних систем.

4.8. Соціальні мережі, рекомендаційні системи, карти — все це графи

Багато цифрових сервісів, якими ми користуємося щодня, від соціальних мереж до навігаторів, працюють на основі єдиної фундаментальної ідеї. Хоча зовні вони здаються абсолютно різними, в їхній основі лежить спільний «креслярський план» — потужна та універсальна структура даних, що називається граф.

Граф є математичною моделлю, що складається з вершин (об'єктів) та ребер (зв'язків між ними). Здатність моделювати не просто об'єкти, а й відношення між ними робить графи ідеальним інструментом для опису складних систем реального світу. Розглянемо, як це працює на конкретних прикладах.

1. Соціальні мережі - граф людських зв'язків

Це, мабуть, найбільш інтуїтивно зрозумілий приклад графа.

- Вершини: Кожен користувач у соціальній мережі — це вершина.
- Ребра: Зв'язок між користувачами — це ребро.

Тип цього графа залежить від самої мережі:

- У Facebook, де дружба є взаємною, це неспрямований граф. Якщо ви друг з кимось, то й він ваш друг.
- У Twitter або Instagram, де ви можете підписатися на когось, а він на вас — ні, це спрямований граф.

Крім того, ребра можуть бути зваженими. Вага ребра може представляти "силу" зв'язку: як часто ви спілкуєтеся, скільки у вас спільних фото, чи реагуєте ви на пости один одного.

Використовуючи алгоритми на графах, соціальні мережі можуть:

- Рекомендувати друзів. Класична логіка "У вас 10 спільних друзів з цією людиною" — це задача пошуку шляхів довжиною 2. Система знаходить всіх ваших друзів (шлях довжиною 1), а потім їхніх друзів (шлях довжиною 2) і рекомендує тих, хто ще не є вашим другом, але до кого існує багато таких коротких шляхів.
- Формувати стрічку новин. Пріоритет надається постам від людей, з якими у вас "сильніший" зв'язок (ребро з більшою вагою).
- Виявляти спільноти та "бульбашки". Алгоритми кластеризації на графах можуть знаходити групи людей (кластери), які щільно пов'язані між собою, але слабко

пов'язані з іншими групами. Це використовується для аналізу поширення інформації, виявлення "ехо-камер" або для цільової реклами.

- Аналізувати впливовість (Influencer Analysis). Люди, які є центральними вузлами з найбільшою кількістю зв'язків (високий ступінь центральності), часто є ключовими фігурами в мережі. Алгоритми можуть ідентифікувати таких «інфлюенсерів» для маркетингових кампаній або аналізу поширення інформації.

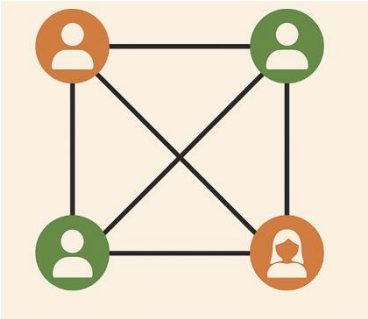


Рис. 24. Граф соціальної мережі

2. Рекомендаційні системи: Граф інтересів

Як Netflix знає, який фільм вам сподобається? Він теж будує граф, але трохи іншого типу. Існують два типи вершин: користувачі та продукти (фільми, товари, пісні). Ребро з'являється між користувачем та продуктом, якщо відбулася взаємодія (перегляд, купівля, висока оцінка). Часто це ребро є зваженим (наприклад, оцінка від 1 до 5).

Такий граф називається дводольним (bipartite), оскільки в ньому вершини чітко розділені на дві групи, і зв'язки існують тільки між вершинами з різних груп (користувач не може бути пов'язаний з іншим користувачем напряму, тільки через спільні інтереси).

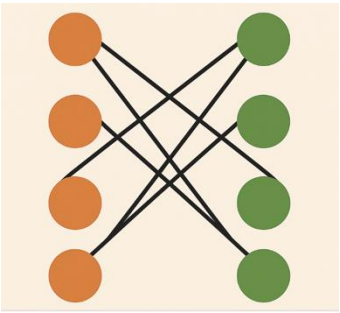


Рис. 25. Дводольний граф

Як працюють рекомендації? Логіка «користувачі, які дивилися цей фільм, також дивляться...» — це задача знаходження закономірностей у цьому графі. Система шукає користувачів зі схожими патернами вподобань (тобто тих, хто має ребра до схожого набору продуктів) і рекомендує вам те, що сподобалося вашим «графовим двійникам». Цей підхід називається колаборативною фільтрацією. Система шукає:

1. Знайди всіх користувачів, пов'язаних ребром з фільмом "Інтерстеллар".

2. Подивись, з якими ще фільмами ці користувачі пов'язані (особливо з високою вагою ребра, тобто високою оцінкою).
3. Якщо багато з цих користувачів також високо оцінили фільм "Прибуття", порекомендуй фільм "Прибуття" початковому користувачу, оскільки їхні смаки, ймовірно, збігаються.

Таким чином, рекомендації — це задача пошуку "непрямих" зв'язків та закономірностей у величезному графі користувачів та продуктів.

3. Карти та навігація: Граф доріг

Коли ви прокладаєте маршрут у Google Maps або Waze, ви працюєте з класичним графом.

- Вершини: Перехрестя, міста або будь-які інші значущі точки на карті.
- Ребра: Дороги, вулиці, авіамаршрути, що з'єднують ці точки.

Це зважений спрямований граф:

- Спрямований, тому що вулиці можуть мати односторонній рух.
- Зважений, тому що кожне ребро (дорога) має "вагу". Це може бути:
  - Відстань (у кілометрах).
  - Час у дорозі (з урахуванням заторів, обмежень швидкості та світлофорів).
  - Вартість (наприклад, для платних доріг).
  - "Мальовничість" маршруту (для туристичних додатків).

Як прокладається маршрут? Це класична задача пошуку найкоротшого шляху (shortest path) у графі. Для її вирішення використовуються класичні алгоритми пошуку шляхів:

- Алгоритм Дейкстри: Гарантовано знаходить найкоротший шлях від однієї вершини до всіх інших, але він досліджує всі можливі напрямки, що може бути повільним на великих картах.
- *Алгоритм A (A-star):** Модифікація алгоритму Дейкстри, яка використовує евристику для більш цілеспрямованого пошуку. На кожному кроці він оцінює не тільки вже пройдену відстань, а й *приблизну* відстань до кінцевої точки (наприклад, по прямій). Це дозволяє йому в першу чергу досліджувати шляхи, що ведуть у правильному напрямку, що робить його значно швидшим.

Навігаційна система перетворює карту на граф і застосовує ці алгоритми, щоб знайти оптимальний маршрут з точки А в точку Б на основі обраного критерію (найшвидший, найкоротший, найдешевший).

Отже, граф виступає як універсальна мова для опису взаємопов'язаного світу. Він дозволяє побачити приховані зв'язки у, на перший погляд, хаотичних даних. Розуміння того, що соціальні мережі, рекомендаційні системи та карти є графами, дозволяє застосовувати до них єдиний, добре вивчений набір алгоритмів для аналізу, пошуку та оптимізації. Це і є сила структур даних: здатність звести складні проблеми реального світу до елегантних математичних моделей, які комп'ютер може ефективно обробляти.

Контрольні запитання до розділу

1. Компроміс масивів та зв'язних списків. Уявіть, що ви розробляєте текстовий редактор. Для зберігання тексту документа ви могли б використати масив символів або двозв'язний список символів. Яку структуру ви б обрали, і чому? Які операції (наприклад, перегляд, вставка/видалення тексту всередині) були б ефективнішими в кожному випадку?
2. Тензор як "надбудова" над масивом. Ми знаємо, що масиви NumPy є надзвичайно ефективними для числових обчислень. Чому тоді фреймворкам для глибокого навчання, як-от PyTorch та TensorFlow, знадобилося створювати власний об'єкт "тензор"? Які дві ключові функціональності, відсутні в NumPy, роблять тензор незамінним для тренування нейронних мереж?
3. Посилання та великі дані в ШІ. Як модель пам'яті Python, що базується на посиланнях (а не на копіюванні об'єктів), допомагає ефективно працювати з величезними датасетами в Pandas або масивами в NumPy? Які потенційні ризики несе такий підхід, якщо розробник недбало змінює дані всередині функцій?
4. Фундаментальна роль хеш-таблиц.і Уявіть, що вам потрібно реалізувати простий лічильник слів для великого тексту без використання словників (dict), наприклад, за допомогою списку кортежів [('слово', 1), ...]. Опишіть, наскільки складним та повільним був би процес оновлення лічильника для кожного нового слова порівняно з використанням хеш-таблиці.
5. Ієрархія проти мережі. У чому полягає принципова різниця між моделюванням даних за допомогою дерева та графа?. Наведіть приклад системи, яку можна представити і як дерево, і як граф (наприклад, корпоративна структура), та поясніть, які аспекти ця система втратить, якщо її спростити від графа до дерева.
6. Наслідки "виродження" дерева. Поясніть своїми словами, чому послідовне додавання вже відсортованих даних у бінарне дерево пошуку є "найгіршим сценарієм". Як це впливає на часову складність операцій і чому збалансовані дерева, як-от АВЛ або червоно-чорні, вирішують цю проблему?
7. Вибір правильної лінійної структури. Наведіть три різні практичні задачі: одну, де ідеальним вибором буде стек (LIFO), другу — де черга (FIFO), і третю — де гнучкість дека (deque) є ключовою перевагою.
8. Піраміда проти збалансованого дерева. І піраміда (Heap), і збалансоване дерево пошуку (BST) є деревними структурами, що гарантують логарифмічну складність для певних операцій. Але вони служать різним цілям. Чому для реалізації черги з пріоритетом (швидкий доступ до мінімального/максимального елемента) використовують саме піраміду, а не збалансоване BST?
9. Граф як модель нейронної мережі. Якщо розглядати нейронну мережу як граф, то що в цьому графі є вершинами, що — ребрами, і яка характеристика ребер є ключовою та змінюється в процесі навчання?
10. Рефлексія про представлення даних. Подумайте про вашу сторінку в соціальній мережі. Спробуйте описати всю інформацію, пов'язану з нею (ваші друзі, пости, лайки, коментарі, групи), використовуючи комбінацію структур даних, вивчених у цій темі. Яку структуру ви б використали для чого і чому?

Тести для самоконтролю

1. У чому полягає основна перевага зв'язних списків над масивами?
    a) Швидший доступ до елемента за індексом (O(1)).
    b) Швидка вставка та видалення елементів у середині списку (O(1)), якщо є вказівник на сусідній вузол.
    c) Менше використання пам'яті.
    d) Можливість зберігати лише елементи одного типу.
2. Ви передали список my_list = [1, 2, 3] у функцію і всередині неї виконали my_list.append(4). Що станеться зі списком поза функцією?
    a) Він залишиться незмінним, [1, 2, 3].
    b) Він зміниться і стане [1, 2, 3, 4], оскільки списки є змінним типом, і у функцію передається посилання на об'єкт.
    c) Виникне помилка, оскільки списки не можна змінювати всередині функцій.
    d) Створиться копія списку, оригінал залишиться незмінним.
3. Яка структура даних працює за принципом LIFO (Last-In, First-Out) і використовується для реалізації стека викликів функцій?
    a) Черга (Queue)
    b) Стек (Stack) c
) Дек (Deque)
    d) Граф
4. Ви моделюєте структуру великої корпорації з чіткою ієрархією (президент, віце-президенти, менеджери, співробітники). Яка структура даних буде найбільш природною для цього?
    a) Граф, оскільки він може моделювати будь-які зв'язки.
    b) Дерево, оскільки воно ідеально представляє ієрархічні відношення "керівник-підлеглий".
    c) Хеш-таблиця, для швидкого доступу до співробітника за його ID.
    d) Стек, для відстеження нещодавно найнятих співробітників.
5. Що є ключовою перевагою хеш-таблиці (dict в Python) для доступу до даних?
    a) Вона зберігає дані у відсортованому порядку.
    b) Вона гарантує відсутність колізій.
    c) Вона забезпечує в середньому дуже швидкий доступ до елемента за ключем (O(1)).
    d) Вона вимагає найменше пам'яті серед усіх структур даних.
6. Яка головна перевага Дерева рішень (Decision Tree) як алгоритму машинного навчання?
    a) Найвища точність серед усіх алгоритмів.
    b) Висока інтерпретованість (легко зрозуміти логіку його рішень).
    c) Стійкість до невеликих змін у даних.
    d) Не потребує великих обсягів даних для навчання.
7. Яка властивість гарантується для Max-Heap (макс-піраміди)?
    a) Усі елементи впорядковані за спаданням.
    b) Найбільший елемент завжди знаходиться у корені дерева.
    c) Кожен лівий нащадок менший за правого.
    d) Дерево завжди є повним (не лише майже повним).
8. Що відбувається з бінарним деревом пошуку, якщо в нього послідовно додавати вже відсортовані дані?
    a) Воно стає ідеально збалансованим.
    b) Воно перетворюється на хеш-таблицю.

c) Воно автоматично видаляє дублікати.

d) Воно вироджується у структуру, схожу на зв'язний список, втрачаючи ефективність.

9. Яка структура даних найкраще підходить для моделювання карти доріг з урахуванням відстаней між містами?

a) Дерево

b) Список списків

c) Зважений граф

d) Хеш-таблиця

10. Чому Python не має вказівників у тому ж сенсі, що й C++?

a) Тому що Python є повільнішою мовою.

b) Тому що Python використовує вищий рівень абстракції — посилання на об'єкти та автоматичне керування пам'яттю (збирач сміття).

c) Тому що в Python неможливо працювати з адресами в пам'яті.

d) Тому що всі змінні в Python є глобальними.

Завдання для самостійної роботи

Це завдання спрямоване на закріплення знань про нелінійні структури даних та їхню роль у моделюванні складних систем, а також на розуміння ключових концепцій керування пам'яттю в Python.

Частина 1: Аналіз та рефлексія (Письмово)

Дайте розгорнуті відповіді на наступні питання.

1. Стек проти Черги. Опишіть фундаментальну різницю між принципами роботи стека (LIFO) та черги (FIFO). Наведіть один практичний приклад (окрім тих, що були в лекції), де використання стека замість черги призвело б до абсолютно некоректної логіки роботи програми.
2. Проблема збалансованості дерев. Поясніть своїми словами, чому послідовне додавання вже відсортованих даних у бінарне дерево пошуку є "найгіршим сценарієм". Як саме ієрархічна структура "ламається" і чому це призводить до деградації продуктивності пошуку з O(log n) до O(n)?
3. Хеш-таблиці та колізії. Хеш-таблиці забезпечують доступ до даних в середньому за час O(1). Однак існує проблема колізій. Поясніть, що таке колізія, і чому, незважаючи на їх існування, хеш-таблиці все одно залишаються ефективними. Опишіть коротко один із методів розв'язання колізій.

Частина 2: Практичне завдання (Програма для моделювання)

Оберіть **одне** із двох завдань та реалізуйте його на Python.

Завдання А: Імітація системи "Undo"

1. Створіть простий імітатор текстового редактора. Використовуйте **стек** (можна реалізувати на базі list або deque) для зберігання станів документа.
2. Створіть початковий порожній рядок-документ.
3. Напишіть цикл, у якому користувач може вводити команди:

- type [текст]: додає введений текст до кінця документа. Попередній стан документа зберігається у стеку.
- undo: скасовує останню зміну, повертаючи документ до попереднього стану зі стека.
- exit: завершує роботу програми.

4. Після кожної команди виводьте на екран поточний стан документа.

Завдання Б: Моделювання простої соціальної мережі

1. Представте невелику соціальну мережу (5-6 користувачів) у вигляді графа. Ви можете використати словник, де ключі — це імена користувачів, а значення — списки їхніх друзів.
2. Напишіть функцію find_mutual_friends(user1, user2), яка приймає імена двох користувачів і повертає список їхніх спільних друзів.
3. Напишіть функцію suggest_friend(user), яка приймає ім'я користувача і пропонує йому в друзі "друга його друзів", з яким у нього найбільше спільних знайомих.

Частина 3: Дослідження та аналіз (Письмово)

Уявіть, що ви проєктуєте рекомендаційну систему для музичного стрімінгового сервісу (наприклад, Spotify).

1. Опишіть, як би ви змоделювали взаємодію користувачів та пісень за допомогою графа. Що у вашому графі було б вершинами, а що — ребрами? Який тип графа (спрямований/неспрямований, зважений/незважений) ви б використали і чому?
2. Як, використовуючи вашу графову модель, можна було б реалізувати функцію "Слухачі цієї пісні також слухають..."? Опишіть логіку пошуку таких рекомендацій на вашому графі.
3. Поміркуйте, як можна було б використати **хеш-таблицю** для кешування результатів роботи вашої рекомендаційної системи, щоб не виконувати складні обчислення на графі щоразу для одного й того ж запиту. Що було б ключем, а що — значенням у вашій хеш-таблиці?

Розділ 5. Алгоритми сортування: наводимо лад у даних

Вступ

Потужність будь-якої моделі штучного інтелекту — від простої регресії до складних нейронних мереж — починається не з самого алгоритму, а з даних. Сирі, необроблені дані часто нагадують хаотичний потік інформації, в якому важко знайти сенс. Перш ніж машина зможе вчитися, ми, як розробники, повинні навести лад у цьому хаосі, і одним з найфундаментальніших інструментів для цього є сортування.

Алгоритми сортування — це не просто академічна вправа з перестановки чисел. У контексті ШІ, хоча ми рідко сортуємо багатомільярдні датасети напряму, принципи впорядкування даних є всюди. Вони використовуються для знаходження медіанних значень при заповненні пропусків, для ранжування елементів у рекомендаційних системах, для ефективного пошуку найближчих сусідів в алгоритмі k-NN, і навіть для візуалізації даних, що допомагає нам зрозуміти їхній розподіл перед навчанням.

У цій темі ми зануримося у світ алгоритмів сортування. Ми розглянемо, як працюють найвідоміші з них — від простих до найефективніших — і зрозуміємо, чому вміння впорядковувати дані є критично важливим першим кроком на шляху до створення інтелектуальних систем.

5.1. Класичні методи сортування

Алгоритми сортування є фундаментальною частиною комп'ютерних наук. Вивчення їхніх внутрішніх механізмів, переваг та недоліків дозволяє зрозуміти ключові компроміси в розробці алгоритмів: між швидкістю, використанням пам'яті, стабільністю та складністю реалізації. Ці знання допомагають не тільки ефективно впорядковувати дані, але й глибше розуміти принципи оптимізації, що лежать в основі багатьох складніших систем.

Категорія 1. Прості квадратичні алгоритми ($O(n^2)$)

Ці алгоритми прості для розуміння, але їхня продуктивність різко падає зі збільшенням розміру даних, що робить їх непридатними для великих наборів. Вони мають переважно освітнє значення, але їхні ідеї можна знайти в більш складних методах.

1. Сортування обміном (бульбашкове сортування) - Bubble Sort

Принцип роботи. Алгоритм багаторазово проходить по масиву. На кожній ітерації він послідовно порівнює кожну пару сусідніх елементів (arr[j] та arr[j+1]) і міняє їх місцями, якщо вони стоять у неправильному порядку. Таким чином, після першого повного проходу найбільший елемент гарантовано "спливає" у самий кінець масиву. Після

другого проходу другий за величиною елемент займає передостанню позицію, і так далі. Цей процес нагадує спливання бульбашок повітря у воді, звідси й назва.

Уявіть ряд людей різного зросту. Ви проходите вздовж ряду і просите кожну пару сусідів, де вищий стоїть ліворуч, помінятися місцями. Повторивши це кілька разів, ви побачите, як найвищі люди поступово "переміститься" в кінець ряду.

Рис.25. Сортування обміном (бульбашкове сортування)

Реалізація. Сортування обміном (бульбашкове сортування) на Python:

```
def bubble_sort(arr):
    n = len(arr)
    # Зовнішній цикл для кожного проходу по масиву
    for i in range(n):
        # Прапор для ранньої зупинки, якщо масив вже відсортовано
        swapped = False
        # Внутрішній цикл для порівняння сусідніх елементів
        # n - i - 1, тому що з кожним проходом останні і елементів вже на своїх місцях
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Класичний обмін елементами
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # Якщо за весь внутрішній цикл не було жодного обміну, масив відсортовано
        if not swapped:
            break
    return arr
```

Часова складність - $O(n^2)$ у середньому та найгіршому випадках (через два вкладені цикли). $O(n)$ у найкращому випадку (якщо масив вже відсортований, і ми використовуємо оптимізацію з прапором `swapped`, що дозволяє завершити роботу після одного проходу).

Просторова складність - $O(1)$, оскільки сортування відбувається "на місці", без використання додаткової пам'яті, пропорційної розміру вхідних даних. Стабільність - стабільний. Якщо два елементи рівні, умова `arr[j] > arr[j + 1]` буде хибною, і вони не поміняються місцями.

2. Сортування вибором (Selection Sort)

Алгоритм ділить масив на дві частини: відсортовану на початку і невідсортовану в кінці. На кожному кроці він знаходить мінімальний елемент у всій невідсортованій частині та міняє його місцями з першим елементом цієї невідсортованої частини. Після цього межа відсортованої частини зсувається на один елемент праворуч.

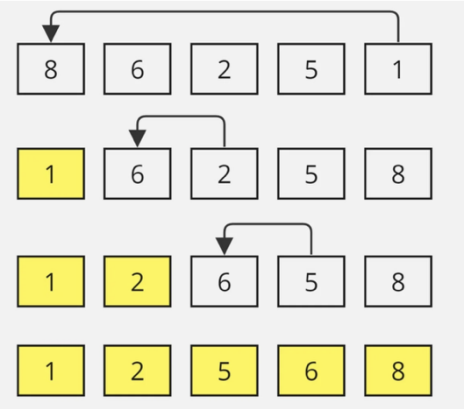


Рис.26. Сортування вибором

Приклад. У вас є ряд людей. Ви знаходите найнижчого і ставите його на перше місце. Потім серед решти (від другої позиції до кінця) знаходите найнижчого і ставите його на друге місце, і так далі, поки всі не будуть впорядковані.

Приклад на Python.

```
def selection_sort(arr):
    n = len(arr)
    # Проходимо по всьому масиву, розширюючи відсортовану частину
    for i in range(n):
        # Припускаємо, що перший елемент невідсортованої частини є мінімальним
        min_idx = i
        # Шукаємо справжній мінімум серед решти елементів
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Міняємо знайдений мінімум з першим елементом невідсортованої частини
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Часова складність - $O(n^2)$. У всіх випадках алгоритм завжди виконує однакову кількість порівнянь, незалежно від початкового стану масиву. Просторова складність - O(1). Стабільність - нестабільний. Обмін віддалених елементів може порушити відносний порядок однакових елементів.

3. Сортування включенням (Insertion Sort)

Алгоритм також ділить масив на відсортовану та невідсортовану частини. Він бере перший елемент з невідсортованої частини і, рухаючись справа наліво по вже

відсортованій частині, "вставляє" його на правильне місце, зсуваючи більші елементи праворуч, щоб "звільнити" для нього місце.

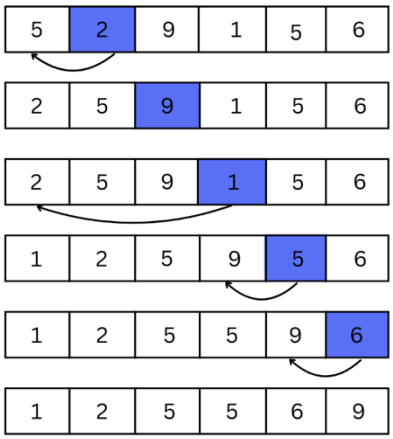


Рис.27. Сортування включенням

Приклад. Сортування карт у руці. Ви берете нову карту з колоди і, порівнюючи її з картами, що вже є у вас в руці (які ви тримаєте впорядкованими), знаходите для неї потрібне місце і вставляєте туди.

Приклад на Python

```
def insertion_sort(arr):
    # Починаємо з другого елемента, оскільки перший вже є "відсортованою" частиною
    for i in range(1, len(arr)):
        key = arr[i] # Елемент, який ми хочемо вставити
        j = i - 1
        # Зсуваємо всі елементи відсортованої частини, що більші за key, праворуч
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        # Вставляємо key на звільнене місце
        arr[j + 1] = key
    return arr
```

Часова складність становитьO(n^2) у середньому та найгіршому випадках. O(n) у найкращому випадку (для майже відсортованих масивів). Це його головна перевага. Просторова складність - O(1). Стабільність - стабільний.

Категорія 2: Ефективні алгоритми (з часовою складністю кращою за O(n^2))

4. Сортування Шелла (Shell Sort)

Це алгоритм, що є значним покращенням сортування включенням. Його часова складність знаходиться між O(n) та O(n^2) і сильно залежить від обраної послідовності кроків. Хоча він не досягає гарантованої складності O(n log n), на практиці він значно швидший за квадратичні алгоритми.

Основна проблема сортування включенням полягає в тому, що елементи можуть переміщатися лише на одну позицію за раз. Якщо найменший елемент знаходиться в кінці масиву, йому доведеться пройти через весь масив. Сортування Шелла вирішує цю проблему, дозволяючи обмін віддалених елементів.

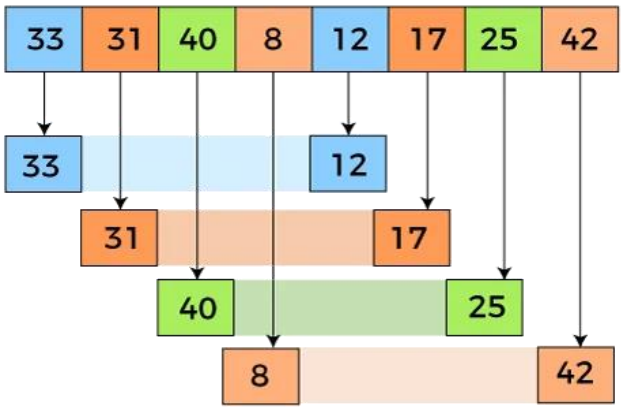


Рис.28. Сортування Шелла

Вибір послідовності кроків (gaps): Спочатку обирається послідовність кроків, яка зменшується і в кінці дорівнює 1 (наприклад, [..., 13, 4, 1]).

1. h-сортування: Для кожного кроку h з послідовності масив сортується за допомогою сортування включенням, але порівнюються не сусідні елементи, а елементи, що знаходяться на відстані h один від одного. Це швидко переміщує елементи ближче до їхніх фінальних позицій.
2. Зменшення кроку: Процес повторюється з меншим кроком.
3. Фінальний крок: Останній крок завжди дорівнює 1. На цьому етапі виконується звичайне сортування включенням, але оскільки масив вже є "майже відсортованим" завдяки попереднім крокам, цей етап виконується дуже швидко (близько до O(n)).

Приклад на Python.

```
def shell_sort(arr):
    n = len(arr)
    # Починаємо з великого кроку, потім зменшуємо його
    gap = n // 2
    while gap > 0:
        # Виконуємо сортування включенням для цього кроку
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

Часова складність залежить від послідовності кроків. Для простого ділення навпіл, як у прикладі, — $O(n^2)$. Для оптимальних послідовностей (наприклад, послідовності Кнута) може досягати O(n^(3/2)) або навіть $O(n \log^2 n)$. Просторова складність - O(1). Стабільність - нестабільний.

5. Сортування злиттям (Merge Sort)

1. Масив рекурсивно ділиться навпіл, поки не залишаться підмасиви з одного елемента (які за визначенням є відсортованими).
2. Відсортовані підмасиви послідовно "зливаються" у більші відсортовані масиви. Процес злиття двох відсортованих масивів є дуже швидким: ми просто порівнюємо перші елементи кожного з них і беремо менший, поки один з масивів не вичерпається.

   Приклад на Python

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Рекурсивний виклик для обох половин
        merge_sort(left_half)
        merge_sort(right_half)

        # Процес злиття
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Копіюємо залишки, якщо вони є
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr
```

Часова складність - O(n log n) у всіх випадках. Це дуже передбачувана і стабільна продуктивність. Просторова складність - O(n), оскільки потребує додаткової пам'яті для тимчасових масивів під час злиття. Це його головний недолік. Стабільність - стабільний.

6. Швидке сортування (Quick Sort)

1. Обирається один елемент з масиву (перший, останній, середній або випадковий).
2. Масив перебудовується таким чином, що всі елементи, менші за опорний, розміщуються ліворуч від нього, а всі більші — праворуч. Після цього кроку опорний елемент знаходиться на своїй фінальній, відсортованій позиції.
3. Алгоритм рекурсивно застосовується до двох підмасивів (ліворуч та праворуч від опорного елемента).

Приклад. Вчитель просить одного учня (опорного) вийти на середину. Потім просить усіх, хто нижчий за нього, стати ліворуч, а хто вищий — праворуч. Тепер цей учень на своєму місці. Після цього вчитель просить двох помічників повторити те саме з лівою та правою групами.

Часова складність алгоритму - O(n log n) у середньому випадку, що робить його одним із найшвидших алгоритмів на практиці. Однак у найгіршому випадку (якщо опорний елемент постійно обирається як найменший або найбільший, наприклад, у вже відсортованому масиві) складність деградує до $O(n^2)$. Просторова складність - O(log n) у середньому випадку (для стека рекурсивних викликів). Стабільність - нестабільний.

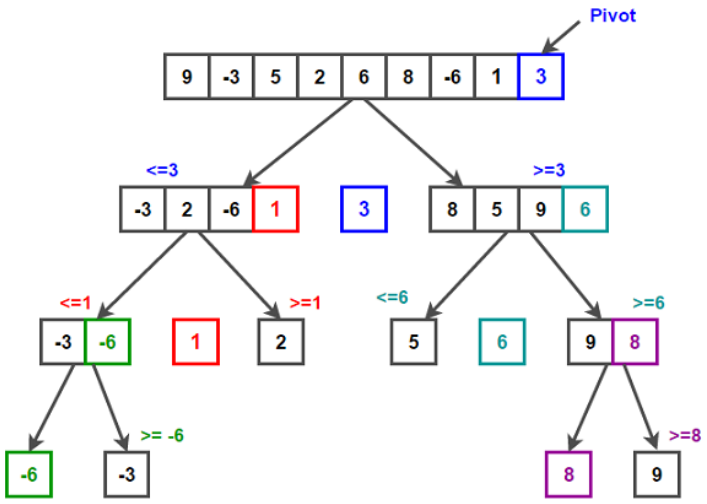


Рис. 29. Швидке сортування (Quick Sort)

Категорія 3: Лінійні алгоритми (не порівняльні)

Ці алгоритми не порівнюють елементи між собою, тому можуть працювати швидше за O(n log n), але лише для специфічних типів даних.

7. Сортування підрахунком (Counting Sort)

Сортування підрахунком (Counting Sort) ефективне лише для сортування цілих невід'ємних чисел у відомому, відносно невеликому діапазоні.

1. Створюється масив-лічильник counts розміром, що дорівнює діапазону значень (k).
2. Проходимо по вхідному масиву і для кожного числа x збільшуємо лічильник counts[x] на одиницю.
3. На основі масиву counts відновлюємо відсортований масив, просто записуючи кожне число стільки разів, скільки вказано в його лічильнику.

Приклад на Python.

```
def counting_sort(arr):
    if not arr: return []
    max_val = max(arr)
    # Створюємо масив-лічильник для значень від 0 до max_val
    counts = [0] * (max_val + 1)
    # Підраховуємо кількість кожного елемента
    for num in arr:
        counts[num] += 1

    sorted_arr = []
    # Відновлюємо відсортований масив
    for i, count in enumerate(counts):
        sorted_arr.extend([i] * count)
    return sorted_arr
```

Часова складність алгоритму - O(n + k), де k — максимальне значення в масиві. Якщо k пропорційне n, складність стає лінійною. Просторова складність - O(k), що може бути проблемою, якщо діапазон значень дуже великий. Стабільність - може бути реалізований як стабільний.

8. Порозрядне сортування (Radix Sort)

Порозрядне сортування сортує числа, розглядаючи їх розряд за розрядом (або байт за байтом).

1. Починаючи з найменш значущого розряду (одиниці), сортує весь масив чисел за цим розрядом. Важливо, щоб сортування на цьому етапі було стабільним (зазвичай використовується сортування підрахунком).
2. Повторює процес для наступного розряду (десятки, сотні і т.д.). Після обробки всіх розрядів масив буде повністю відсортованим.

Приклад на Python (з використанням Counting Sort як стабільного підсортування):

```
def counting_sort_for_radix(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    # Підраховуємо кількість входжень цифр
    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1
```

```
    # Змінюємо count[i] так, щоб він містив позицію цифри в output
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Будуємо вихідний масив
    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    # Копіюємо результат
    for i in range(len(arr)):
        arr[i] = output[i]

def radix_sort(arr):
    if not arr: return []
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10
    return arr
```

Часова складність алгоритму - O(d * (n + k)), де d — кількість розрядів, n — кількість елементів, k — основа системи числення (10 для десяткових). Просторова складність - O(n + k). Стабільність залежить від стабільності підсортування (у нашому випадку, стабільне).

Отже, не існує єдиного "найкращого" алгоритму сортування — вибір завжди залежить від конкретної задачі, розміру даних, їхнього початкового стану та вимог до стабільності та пам'яті.

- Квадратичні алгоритми варто використовувати лише для навчальних цілей або для дуже маленьких масивів. Сортування включенням є винятком, оскільки його ефективність на майже відсортованих даних робить його корисним у гібридних алгоритмах.
- Алгоритми O(n log n) є стандартним вибором для загальноцільового сортування. Merge Sort є надійним та передбачуваним, але вимагає додаткової пам'яті. Quick Sort зазвичай швидший на практиці, але має потенційно погану продуктивність у найгіршому випадку.
- Лінійні алгоритми є надзвичайно потужними, але вузькоспеціалізованими. Вони ідеальні, коли ви працюєте з цілими числами в обмеженому діапазоні.

Розуміння цих компромісів є ключовим. Наприклад, вбудований у Python алгоритм сортування Timsort є гібридним: він використовує Merge Sort для розділення великого масиву на частини та Insertion Sort для ефективного сортування цих невеликих частин. Це дозволяє йому поєднувати найкращі риси обох підходів.

5.2. Контекст ШІ: чому сортування важливе?

Поширена думка, що в епоху складних нейронних мереж, здатних обробляти хаотичні дані, класичні алгоритми сортування втратили свою актуальність. Однак такий погляд ігнорує фундаментальну роль впорядкування даних у всьому життєвому циклі розробки систем ШІ. Хоча ми рідко застосовуємо сортування до всього датасету напряму, впорядкування даних є фундаментальною операцією, яка незримо присутня на багатьох етапах роботи зі штучним інтелектом.

Сортування — це не самоціль, а потужний інструмент, що робить дані зрозумілішими, а алгоритми — ефективнішими. Розглянемо ключові сфери, де його роль є критично важливою.

Попередня обробка даних (Data Preprocessing)

Це етап, де готується "паливо" для моделі. Якість цього етапу безпосередньо впливає на кінцевий результат, і сортування тут відіграє ключову роль. Заповнення пропущених значень (Imputation), один з найпоширеніших методів заповнення пропусків у числових ознаках — це використання медіани. Щоб знайти медіану, потрібно спочатку відсортувати всі наявні значення і взяти те, що знаходиться посередині. Медіана є більш стійкою до викидів, ніж середнє арифметичне, тому сортування для її обчислення є стандартною практикою.

Через сортування відбувається виявлення та обробка викидів (Outlier Detection). Коли дані відсортовані, аномально великі або малі значення опиняються на краях масиву. Це дозволяє легко їх ідентифікувати за допомогою процентилів (наприклад, все, що вище 99-го процентиля). Метод квантильного перетворення (Quantile Transformation) використовується для перетворення розподілу ознаки на рівномірний. Для цього потрібно відсортувати дані та розділити їх на рівні частини (квантилі, децилі, процентилі). Для побудови багатьох важливих графіків, як-от кумулятивна функція розподілу (CDF) або квантиль-квантиль (Q-Q) плот, дані спочатку необхідно відсортувати а потім візуалізувати. Це дозволяє візуально оцінити розподіл ознаки та порівняти його з еталонним.

Внутрішня робота алгоритмів машинного навчання

Багато алгоритмів МН "під капотом" використовують принципи сортування для своєї роботи. Існує простий, але потужний алгоритм класифікації та регресії -

k-Найближчих Сусідів (k-Nearest Neighbors, k-NN). Щоб зробити прогноз для нової точки, він:

1. Обчислює відстань від цієї точки до всіх точок у навчальному наборі.
2. Сортує ці відстані за зростанням.
3. Обирає k точок з найменшими відстанями (найближчих сусідів) і робить прогноз на основі їхніх класів або значень. Крок сортування тут є центральним.

Для розділення даних за певними ознаками використовують Дерева рішень (Decision Trees). Для числових ознак один з ефективних способів знайти оптимальний поріг — це відсортувати всі унікальні значення ознаки і перевірити кожен з них як потенційний кандидат для розділення.

Ранжування та Рекомендаційні системи

У цій сфері сортування є не просто допоміжною операцією, а кінцевим продуктом.

Коли ви вводите запит у Google, система знаходить мільйони релевантних сторінок, а потім сортує їх за сотнями факторів (релевантність, авторитетність, свіжість), щоб показати вам на першій сторінці найкращі результати. Сортування використовують рекомендаційні та пошукові системи (Netflix, Spotify, Amazon), що видають список результатів. Модель ШІ обчислює «оцінку релевантності» для кожного продукту (фільму, пісні, товару), а потім сортує їх за цією оцінкою. Це стосується як персоналізованих стрічок, так і результатів внутрішнього пошуку на платформі.

В HR-системах ШІ може оцінювати резюме кандидатів за певною шкалою, а потім ранжувати (сортувати) їх, щоб рекрутер міг в першу чергу звернути увагу на найбільш відповідних.

Приклади використання конкретних алгоритмів сортування в ШІ

Хоча часто ми не задумуємось, який саме алгоритм використовується у функції sort(), розуміння їхніх властивостей дозволяє вирішувати специфічні задачі.

Стабільне сортування (наприклад, Сортування злиттям - Merge Sort) використовується у багаторівневому ранжуванні результатів. Уявіть інтернет-магазин. Ви хочете відсортувати товари за рейтингом покупців (від 5 зірок до 1). Але серед товарів з однаковим рейтингом ви хочете, щоб більш популярні (ті, що частіше купують) стояли вище. Ви можете спочатку відсортувати всі товари за популярністю, а потім виконати **стабільне** сортування за рейтингом. Стабільність гарантує, що якщо два товари мають однаковий рейтинг (наприклад, 4.5 зірки), їхній відносний порядок, встановлений на першому етапі (за популярністю), не зміниться. Merge Sort і Timsort (використовується в Python) є стабільними.

Сортування підрахунком (Counting Sort) використовується для обробки зображень, аналізу категоріальних даних. Наприклад, у комп'ютерному зорі для багатьох завдань (як-от, для вирівнювання контрасту) потрібно побудувати гістограму інтенсивності пікселів. Оскільки яскравість кожного пікселя в чорно-білому зображенні — це ціле число від 0 до 255, діапазон значень є дуже малим. Сортування підрахунком дозволяє отримати відсортований масив інтенсивностей або їхню частотну гістограму за лінійний час O(n + k), де k=256, що значно швидше за будь-який алгоритм, заснований на порівняннях (O(n log n)).

Пірамідальне сортування (Heapsort) та структури даних "піраміда" використовуються для пошуку k-найкращих елементів без повного сортування. Наприклад, Netflix не потрібно сортувати всі мільйони фільмів для вас. Йому достатньо показати топ-10, які вам, імовірно, сподобаються. Для цього можна використати структуру Min-Heap (мін-піраміда) розміром 10. Алгоритм проходить по всіх фільмах. Для кожного фільму: якщо його прогнозований рейтинг вищий, ніж найменший рейтинг у піраміді (який завжди знаходиться у її корені), то цей найменший елемент видаляється, а на його місце вставляється новий, більш релевантний фільм. Після обробки всіх фільмів у піраміді залишаться 10 найкращих. Це значно ефективніше (складність O(N log k)), ніж сортувати весь набір (складність O(N log N)).

Оцінка та інтерпретація моделей

Навіть після того, як модель навчена, сортування допомагає нам оцінити її якість.

Для оцінки якості бінарного класифікатора використовується ROC-крива. Щоб її побудувати, потрібно відсортувати всі прогнози моделі за спаданням їхньої ймовірності, а потім послідовно змінювати поріг класифікації, обчислюючи метрики True Positive Rate та False Positive Rate. Аналогічний підхід, що базується на сортуванні прогнозів,

використовується і для побудови інших важливих діаграм, таких як Precision-Recall крива та Lift Chart (діаграма підйому), які допомагають оцінити бізнес-цінність моделі.

Таким чином, хоча виклик data.sort() до всього датасету є рідкістю, сортування виступає невидимим, але фундаментальним процесом, що перетворює хаотичні дані та сирі прогнози на структуровані, інтерпретовані та корисні для бізнесу результати. Воно дозволяє нам готувати дані, розуміти їхню структуру, є невід'ємною частиною роботи багатьох ключових алгоритмів та, що найважливіше, допомагає перетворити сирі прогнози моделей на корисний, впорядкований та зрозумілий для кінцевого користувача продукт. Це фундаментальна операція, що привносить порядок у світ даних.

Контрольні запитання до розділу

1. *Практичний вибір квадратичних алгоритмів.* Хоча сортування включенням, вибором та бульбашкове мають однакову середню складність $O(n^2)$, сортування включенням часто вважається найкращим серед них для невеликих масивів. Чому, на вашу думку, це так? Яка особливість його роботи робить його ефективним на майже відсортованих даних?
2. *Ціна стабільності.* Сортування злиттям (Merge Sort) є стабільним, а швидке сортування (Quick Sort) — ні. Наведіть конкретний приклад задачі (окрім сортування товарів у магазині), де збереження відносного порядку однакових елементів є критично важливим, і використання нестабільного алгоритму призвело б до некоректного результату.
3. *Обмеження лінійних алгоритмів.* Сортування підрахунком (Counting Sort) та порозрядне сортування (Radix Sort) можуть працювати за лінійний час, що значно швидше за O(n log n). Чому ж тоді ми не використовуємо їх завжди? Які фундаментальні обмеження цих алгоритмів не дозволяють застосовувати їх, наприклад, для сортування списку імен або дійсних чисел?
4. *"Найгірший випадок" Quick Sort*. Найгірший випадок для швидкого сортування ($O(n^2)$) виникає, коли опорний елемент (pivot) постійно обирається як найменший або найбільший. Як можна модифікувати алгоритм вибору опорного елемента, щоб значно зменшити ймовірність настання цього найгіршого випадку на практиці?
5. *n-place vs. Out-of-place*. Пірамідальне сортування (Heapsort) та сортування злиттям (Merge Sort) мають однакову часову складність O(n log n). Однак один з них є "in-place" (вимагає O(1) додаткової пам'яті), а інший — "out-of-place" (вимагає O(n) додаткової пам'яті). У якому практичному сценарії, пов'язаному з обробкою великих даних, ця різниця у вимогах до пам'яті буде вирішальною?
6. *Роль сортування в k-NN.* Алгоритм k-найближчих сусідів робить прогноз на основі "голосування" k найближчих точок. Очевидно, що для цього потрібно відсортувати відстані. Як зміниться складність цього алгоритму, якщо замість повного сортування всіх відстаней ми використаємо більш оптимальний підхід, наприклад, піраміду, щоб знайти лише k найменших відстаней?
7. *Сортування та попередня обробка даних.* Опишіть, як операція сортування допомагає вирішити задачу виявлення та обробки викидів (outliers) у наборі числових даних. Які метрики (наприклад, процентилі) стає легко обчислити після сортування?
8. *Гібридні алгоритми.* Вбудований у Python алгоритм Timsort є гібридним. Він поєднує ідеї сортування злиттям та сортування включенням. Поясніть логіку такого поєднання.

Чому не використовується лише один з цих алгоритмів, а саме їхня комбінація робить Timsort настільки ефективним на практиці?

9. *Поза числовими даними.* Ми звикли говорити про сортування чисел. Уявіть, що вам потрібно відсортувати набір складних об'єктів (наприклад, профілі користувачів) за кількома критеріями одночасно (наприклад, спочатку за датою реєстрації, а потім за кількістю активності). Як концепція "ключа сортування" та стабільність алгоритму допомагають вирішити цю задачу?
10. *Рефлексія про компроміси.* Не існує єдиного "найкращого" алгоритму сортування. Кожен з них є компромісом між часовою складністю (у середньому та найгіршому випадках), просторовою складністю та стабільністю. Наведіть приклад задачі, для якої ви б обрали Quick Sort, і приклад, де ви б категорично віддали перевагу Merge Sort, обґрунтувавши свій вибір.

Тести для самоконтролю

1. Який алгоритм сортування є прикладом стратегії "Розділяй і володарюй" і має гарантовану часову складність O(n log n) у всіх випадках?
    a) Швидке сортування (Quick Sort)
    b) Сортування включенням (Insertion Sort)
    c) Сортування злиттям (Merge Sort)
    d) Бульбашкове сортування (Bubble Sort)
2. У чому полягає основний недолік сортування злиттям (Merge Sort) порівняно з пірамідальним сортуванням (Heapsort)?
    a) Його часова складність у найгіршому випадку — $O(n^2)$.
    b) Він вимагає O(n) додаткової пам'яті, тоді як Heapsort працює "на місці" (O(1) пам'яті).
    c) Він не є стабільним алгоритмом.
    d) Він значно повільніший на практиці.
3. Яка з цих ситуацій є "найкращим випадком" для сортування включенням (Insertion Sort), де його складність буде O(n)?
    a) Масив відсортований у зворотному порядку.
    b) Масив заповнений випадковими числами.
    c) Масив вже майже відсортований.
    d) Масив містить лише унікальні елементи.
4. Сортування підрахунком (Counting Sort) буде надзвичайно ефективним для:
    a) Сортування списку імен.
    b) Сортування дійсних чисел від 0.0 до 1.0.
    c) Сортування масиву, що містить інтенсивність пікселів (цілі числа від 0 до 255).
    d) Сортування великих чисел, що мають до 100 розрядів.
5. Що означає, що алгоритм сортування, такий як сортування злиттям, є "стабільним"?
    a) Він завжди працює з однаковою швидкістю.
    b) Він не змінює відносний порядок елементів з однаковими значеннями.
    c) Він стійкий до помилок у вхідних даних.
    d) Він працює без використання рекурсії.
6. Чому сортування є важливим кроком в алгоритмі k-найближчих сусідів (k-NN)?
    a) Воно допомагає нормалізувати дані.

b) Воно дозволяє впорядкувати всіх сусідів за відстанню, щоб легко обрати k найближчих.
c) Воно робить обчислення відстані швидшим.
d) Воно є обов'язковим для будь-якого алгоритму класифікації.

7. Який алгоритм сортування на кожному кроці знаходить найменший елемент у невідсортованій частині масиву і переміщує його на початок?
a) Бульбашкове сортування
b) Сортування включенням
c) Сортування вибором (Selection Sort)
d) Сортування Шелла

8. Найгірший випадок для Швидкого сортування (Quick Sort), що призводить до складності $O(n^2)$, виникає, коли:
a) Масив містить багато дублікатів.
b) Масив заповнений випадковими даними.
c) Масив вже відсортований (або відсортований у зворотному порядку), і опорний елемент постійно обирається як перший або останній.
d) Масив має непарну кількість елементів.

9. Який з цих алгоритмів НЕ є алгоритмом, заснованим на порівняннях?
a) Сортування вибором
b) Пірамідальне сортування
c) Сортування злиттям
d) Порозрядне сортування (Radix Sort)

10. Пірамідальне сортування (Heapsort) та Швидке сортування (Quick Sort) мають однакову середню часову складність $O(n \log n)$. У якому випадку Heapsort матиме явну перевагу над Quick Sort?
a) Коли потрібно відсортувати невеликий масив.
b) Коли потрібен стабільний алгоритм сортування.
c) Коли критично важливо гарантувати продуктивність $O(n \log n)$ навіть у найгіршому випадку.
d) Коли потрібно сортувати рядки, а не числа.

Завдання для самостійної роботи

Це завдання допоможе вам глибше зрозуміти принципи роботи класичних алгоритмів сортування, їхні компроміси та практичне застосування в контексті штучного інтелекту.

Частина 1: Аналіз та рефлексія (Письмово)

Дайте розгорнуті відповіді на наступні питання.

1. Компроміс Quick Sort vs. Merge Sort. Обидва алгоритми мають середню часову складність $O(n \log n)$. Однак, швидке сортування (Quick Sort) часто швидше на практиці, але має найгірший випадок $O(n^2)$, тоді як сортування злиттям (Merge Sort) завжди гарантує $O(n \log n)$, але вимагає додаткової пам'яті. Поясніть, у якому сценарії ви б обрали один алгоритм, а в якому — інший, і чому цей вибір є важливим.
2. Сила та слабкість лінійних алгоритмів. Сортування підрахунком (Counting Sort) та порозрядне сортування (Radix Sort) можуть працювати значно швидше за

алгоритми, засновані на порівняннях. У чому полягає їхнє головне обмеження, яке не дозволяє використовувати їх для сортування, наприклад, списку рядків або дійсних чисел? Наведіть приклад задачі в ШІ, де такі обмеження не є проблемою, і використання цих алгоритмів буде виправданим.

3. Стабільність сортування. Поясніть своїми словами, що таке "стабільність" алгоритму сортування. Уявіть, що ви працюєте з таблицею даних про співробітників, яку потрібно відсортувати спочатку за назвою відділу (в алфавітному порядку), а потім — за зарплатою (за спаданням). Чому важливо, щоб другий етап сортування (за зарплатою) виконувався нестабільним алгоритмом, якщо ви хочете, щоб співробітники з однаковою зарплатою залишилися згрупованими за відділами?

Частина 2: Практичне завдання (Реалізація та порівняння)

Напишіть скрипт на Python, який виконує наступні дії:

1. Реалізуйте два алгоритми сортування з нуля:
   - Сортування включенням (Insertion Sort): як приклад простого квадратичного алгоритму.
   - Сортування злиттям (Merge Sort): як приклад ефективного рекурсивного алгоритму.

   Не використовуйте вбудовані функції сортування.
2. Створіть тестові дані:
   - Згенеруйте три масиви з 5000 випадкових цілих чисел:
     - Перший — повністю випадковий.
     - Другий — вже відсортований.
     - Третій — відсортований у зворотному порядку.
3. Проведіть експеримент:
   - Для кожного з трьох масивів виміряйте час виконання обох реалізованих вами алгоритмів сортування. Використовуйте модуль time для вимірювання.
   - Виведіть результати у вигляді таблиці або зрозумілого текстового звіту.
4. Зробіть висновки:

   У коментарях до коду або в окремому текстовому блоці поясніть отримані результати. Чи відповідають вони теоретичним очікуванням щодо часової складності цих алгоритмів у найкращому, середньому та найгіршому випадках?

Частина 3: Дослідження та аналіз (Письмово)

У лекції згадувалося, що сортування є ключовим етапом в алгоритмі k-найближчих сусідів (k-NN). Однак повне сортування всіх відстаней може бути надлишковим, якщо нам потрібно знайти лише невелику кількість k сусідів.

1. Опишіть стандартний алгоритм k-NN, акцентуючи увагу на етапі, де використовується сортування. Яка буде загальна часова складність цього етапу, якщо використовувати ефективний алгоритм сортування (наприклад, Timsort)?
2. Дослідіть та опишіть альтернативний підхід до знаходження k найближчих сусідів, який не вимагає повного сортування всього масиву відстаней. (Підказка: подумайте про використання структури даних, яка ефективно знаходить та вилучає мінімальний або максимальний елемент, наприклад, піраміда (Heap)).
3. Порівняйте часову складність стандартного підходу (з повним сортуванням) та оптимізованого підходу (з використанням піраміди). Чому другий підхід є значно ефективнішим, особливо коли k набагато менше за загальну кількість даних n?

Розділ 6. Алгоритми пошуку: від пошуку в масиві до пошуку шляху

Вступ

Пошук — це, мабуть, одна з найчастіших дій, які ми виконуємо у цифровому світі. Ми шукаємо інформацію в Google, потрібний файл на комп'ютері, друга у списку контактів або найкоротший шлях додому на карті. В основі всіх цих, на перший погляд, різних завдань лежать алгоритми пошуку — фундаментальні інструменти, що дозволяють комп'ютеру ефективно знаходити потрібні дані у величезних масивах інформації.

Але "знайти" можна по-різному. Стратегія пошуку кардинально змінюється залежно від того, як організовані наші дані. Знайти слово у невпорядкованому списку — це одне, а знайти його у відсортованому словнику — зовсім інше. Ще складніше завдання — знайти не просто один елемент, а цілу послідовність кроків, або **шлях**, що веде до мети.

У цій темі ми пройдемо шлях від найпростіших до найскладніших ідей у світі пошуку. Ми почнемо з базових методів пошуку в масивах, потім розглянемо, як сортування даних дозволяє досягти неймовірного приросту у швидкості, і, нарешті, перейдемо до найцікавішого — алгоритмів пошуку шляху в графах, які є основою для GPS-навігаторів, ігрового штучного інтелекту та багатьох інших сучасних технологій.

6.1. Класичні алгоритми пошуку в масивах: лінійний, бінарний, інтерполяційний

Пошук елемента в масиві даних є однією з найбільш фундаментальних операцій у програмуванні. Вибір правильного алгоритму пошуку залежить від ключової характеристики даних: чи є вони впорядкованими. Від цього вибору безпосередньо залежить ефективність та швидкість роботи програми, особливо при обробці великих обсягів інформації.

Розглянемо три класичні підходи до пошуку в масивах, від найпростішого до більш інтелектуального.

1. Лінійний пошук (Linear Search)

Лінійний пошук (Linear Search) - найпростіший та найбільш інтуїтивно зрозумілий алгоритм. Принцип роботи наступний. Послідовно перевіряти кожен елемент масиву, починаючи з першого, доки не буде знайдено потрібний елемент або поки не закінчаться всі елементи. Вимоги до даних відсутні. Лінійний пошук працює як на відсортованих, так і на невпорядкованих даних. Це його головна перевага. Прикладом лінійного пошуку може бути пошук потрібної книги на полиці, де всі книги розставлені у випадковому порядку. Вам доведеться переглядати їх одну за одною.

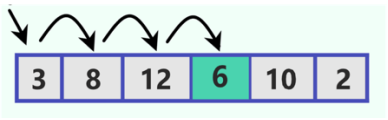


Рис.30. Приклад лінійного пошуку числа 6 в масиві

Приклад на Python:

```
def linear_search(arr, target):
    """
    Виконує лінійний пошук елемента target у масиві arr.
    Повертає індекс елемента, якщо знайдено, інакше -1.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Елемент знайдено
    return -1 # Елемент не знайдено

# Приклад використання на невпорядкованих даних
data = [3, 8, 12, 6, 10, 2]
target_element = 6
index = linear_search(data, target_element)
print(f"Лінійний пошук: елемент {target_element} знайдено за індексом {index}")
```

Часова складність алгоритму - O(1) у найкращому випадку, коли шуканий елемент є першим у масиві. у середньому та найгіршому випадках - O(n), коли потрібно перевірити половину або всі елементи масиву відповідно.

2. Бінарний пошук (Binary Search)

Бінарний пошук (Binary Search) - значно більш ефективний алгоритм, але він має одну критично важливу вимогу. Алгоритм порівнює шуканий елемент із середнім елементом масиву.

- Якщо вони рівні, пошук завершено.
- Якщо шуканий елемент менший за середній, пошук продовжується лише в лівій половині масиву.
- Якщо шуканий елемент більший за середній, пошук продовжується лише в правій половині масиву.

Цей процес повторюється, на кожному кроці відкидаючи половину даних, що залишилися.

Масив обов'язково має бути відсортованим. Прикладом алгоритму бінарного пошуку може бути пошук слова у паперовому словнику. Ви відкриваєте його приблизно посередині. Якщо шукане слово йде за алфавітом раніше, ви відкидаєте другу половину словника і продовжуєте пошук у першій.

Приклад на Python (ітеративний підхід):

```
def binary_search(sorted_arr, target):
    """
    Виконує бінарний пошук у відсортованому масиві.
```

```
    Повертає індекс елемента, якщо знайдено, інакше -1.
    """
    low = 0
    high = len(sorted_arr) - 1

    while low <= high:
        mid = (low + high) // 2
        guess = sorted_arr[mid]

        if guess == target:
            return mid
        elif guess > target:
            high = mid - 1 # Відкидаємо праву половину
        else:
            low = mid + 1  # Відкидаємо ліву половину
    return -1

# Приклад використання на відсортованих даних
sorted_data = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
target_element = 23
index = binary_search(sorted_data, target_element)
print(f"Бінарний пошук: елемент {target_element} знайдено за індексом {index}")
```

На кожному кроці ми вдвічі зменшуємо простір для пошуку, тому часова складність - **O(log n)**. Для масиву з мільйона елементів знадобиться не більше 20 порівнянь.

3. Інтерполяційний пошук (Interpolation Search)

Інтерполяційний пошук (Interpolation Search) - це покращена версія бінарного пошуку, яка намагається бути "розумнішою". Замість того, щоб завжди перевіряти середній елемент, інтерполяційний пошук робить "освічене припущення" про те, де може знаходитися шуканий елемент. Він враховує не тільки індекси початку та кінця, але й значення на цих позиціях. Якщо ви шукаєте маленьке число у діапазоні від 1 до 1000, він буде шукати ближче до початку масиву, а не посередині. Масив має бути відсортованим, а його значення — розподілені приблизно рівномірно. Це означає, що різниця між сусідніми елементами є більш-менш сталою (як в арифметичній прогресії). Наприклад, масив [10, 20, 30, 40] є ідеальним, тоді як [1, 2, 100, 101] — поганим, оскільки розподіл значень нерівномірний.

Як приклад - пошук прізвища на літеру "Т" у телефонній книзі. Ви не відкриєте її посередині (на літеру "М"), а одразу перейдете значно ближче до кінця.

Приклад на Python:

```
def interpolation_search(sorted_arr, target):
    """
    Виконує інтерполяційний пошук у відсортованому, рівномірно розподіленому
масиві.
    """
    low = 0
    high = len(sorted_arr) - 1

    while low <= high and sorted_arr[low] <= target <= sorted_arr[high]:
```

```
        # Уникаємо ділення на нуль
        if sorted_arr[low] == sorted_arr[high]:
            if sorted_arr[low] == target:
                return low
            return -1
        # Формула для прогнозування позиції
        pos = low + ((target - sorted_arr[low]) * (high - low)) // (sorted_arr[high] -
sorted_arr[low])

        if sorted_arr[pos] == target:
            return pos
        elif sorted_arr[pos] < target:
            low = pos + 1
        else:
            high = pos - 1
    return -1

# Приклад використання на рівномірно розподілених даних
uniform_data = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
target_element = 80
index = interpolation_search(uniform_data, target_element)
print(f"Інтерполяційний пошук: елемент {target_element} знайдено за індексом
{index}")
```

У середньому випадку (на рівномірно розподілених даних) — часова складність становитиме **O(log log n)**, що навіть швидше за бінарний пошук. Однак у найгіршому випадку (наприклад, на експоненціально зростаючих даних) складність може деградувати до **O(n)**.

Хоча ми рідко реалізуємо ці алгоритми вручну, використовуючи оптимізовані методи бібліотек, їхні принципи лежать в основі багатьох процесів у машинному навчанні.

Лінійний пошук застосовується для перевірки невеликих, невпорядкованих наборів або умов, наприклад, у задачах NLP (обробки природної мови) часто використовується список стоп-слів (наприклад, ['і', 'в', 'на', 'але', 'це']). Коли програма обробляє текст, вона для кожного слова перевіряє, чи не входить воно до цього списку. Оскільки такий список зазвичай невеликий, для цієї перевірки "під капотом" виконується ефективний лінійний пошук.

Бінарний пошук в ШІ є центральним для багатьох оптимізаційних задач, зокрема, для оптимізація гіперпараметрів. Уявіть, що ви налаштовуєте гіперпараметр регуляризації alpha для моделі Ridge-регресії. Ви знаєте, що оптимальне значення лежить десь у діапазоні [0.001, 100.0]. Замість того, щоб перебирати десятки значень, можна застосувати метод, схожий на бінарний пошук: протестувати середнє значення, оцінити якість моделі, і залежно від результату (модель перенавчена чи недонавчена) відкинути половину діапазону та продовжити пошук у частині, що залишилася.
Бінарний пошук також використовується для пошуку оптимального порогу класифікації. наприклад, модель для визначення спаму видає ймовірність від 0 до 1. Нам потрібно обрати поріг (наприклад, 0.7), вище якого лист буде вважатися спамом. Щоб знайти найкращий поріг, який максимізує метрику F1-score, ми можемо перевірити

значення у відсортованому діапазоні [0.0, 1.0]. Бінарний пошук дозволить швидко знайти поріг, що забезпечує найкращий баланс між точністю та повнотою.
Інтерполяційний пошук в ШІ використовується для роботи з даними, що мають рівномірний розподіл. Прикладом може бути аналіз часових рядів з сенсорів, які записують дані через однакові проміжки часу (наприклад, кожну секунду). Якщо нам потрібно знайти запис, що відповідає конкретній секунді, інтерполяційний пошук буде значно ефективнішим за бінарний. Він може майже точно "вгадати" індекс у масиві, виходячи з того, що дані розподілені лінійно.

***Таблиця 18**. Загальні характеристики алгоритмів пошуку*

Алгоритм	**Часова складність (середня)**	**Вимоги до даних**	**Переваги**	**Недоліки**
Лінійний	O(n)	Жодних	Простота, працює на будь-яких даних	Дуже повільний для великих масивів
Бінарний	O(log n)	Відсортовані дані	Дуже швидкий та надійний	Потребує попереднього сортування
Інтерполяційний	O(log log n)	Відсортовані та рівномірно розподілені	Ще швидший за бінарний в ідеальних умовах	Чутливий до розподілу даних

Отже, вибір алгоритму пошуку — це завжди аналіз компромісу між вартістю підготовки даних та швидкістю доступу до них. Лінійний пошук не вимагає жодної підготовки, але платить за це повільною роботою. Бінарний та інтерполяційний пошуки є надзвичайно швидкими, але вимагають значних початкових витрат на сортування даних (зазвичай O(n log n)). Тому їхнє використання виправдане лише тоді, коли пошук у наборі даних виконується багаторазово, дозволяючи амортизувати вартість сортування.

6.2. Акцент на ШІ: введення у поняття пошуку у просторі станів

Коли ми говоримо про пошук, ми часто уявляємо знаходження одного елемента в масиві або списку. Однак у світі штучного інтелекту завдання рідко бувають такими простими. Як знайти найкращий маршрут з точки А в точку Б? Як розв'язати головоломку "п'ятнашки"? Як знайти виграшну послідовність ходів у шахах?

Всі ці, на перший погляд, різні проблеми об'єднує спільна ідея: вони можуть бути представлені як задача пошуку у просторі станів (State Space Search). Це одна з найстаріших і найважливіших концепцій у класичному ШІ, яка дозволяє звести складну проблему до задачі пошуку шляху в графі. Це фундаментальний перехід від питання «де знаходиться відповідь?» до питання «як отримати відповідь?», тобто від пошуку статичної інформації до знаходження послідовності дій.

Щоб зрозуміти цей підхід, потрібно визначити кілька ключових термінів:

1. Стан (State) - це унікальна конфігурація або "знімок" задачі в певний момент часу. Важливо, що стан має містити всю необхідну інформацію для прийняття подальших рішень.
   - У GPS-навігаторі стан — це ваше поточне місцезнаходження (наприклад, конкретне перехрестя).
   - У грі в шахи стан — це не тільки поточне розташування всіх фігур на дошці, але й чий зараз хід, чи доступні рокіровки, і чи можливе взяття на проході.
   - У головоломці "п'ятнашки" стан — це поточне розташування всіх плиток.
2. Простір станів (State Space) - це вся множина всіх можливих станів, у яких може перебувати задача. Це, по суті, повна "карта" всіх можливих конфігурацій. Розмір простору станів є критично важливим. Для головоломки 3x3 він є відносно невеликим, але для шахів кількість можливих станів перевищує кількість атомів у видимому Всесвіті. Це явище, відоме як комбінаторний вибух, робить неможливим повний перебір всіх варіантів навіть для найпотужніших комп'ютерів. Саме тому необхідні 'розумні' алгоритми пошуку, здатні відсікати безперспективні шляхи, не досліджуючи їх.
3. Початковий стан (Initial State) - стан, з якого починається пошук.
   - У GPS — ваше поточне місцезнаходження.
   - У грі — початкове розташування фігур.
4. Цільовий стан (Goal State) - стан (або множина станів), який є розв'язком задачі.
   - У GPS — пункт призначення.
   - У головоломці — зібрана картинка.
   - У шахах — будь-який стан, де супернику поставлено мат.
5. Оператори (або дії, Actions) - це правила, що дозволяють переходити з одного стану в інший. Кожен оператор може мати свою "вартість" (cost). Наприклад, у GPS вартість — це відстань або час. У головоломці вартість кожного ходу зазвичай дорівнює 1.

Найприродніший спосіб представити простір станів — це граф. Стани є вершинами (вузлами) графа. Оператори (дії) є ребрами, що з'єднують ці вершини. Таким чином, будь-яка задача, яку можна описати в термінах станів та переходів між ними, автоматично перетворюється на задачу пошуку в графі. Важливо розуміти, що цей граф зазвичай є неявним (implicit). Тобто ми не створюємо і не зберігаємо його повністю в пам'яті. Натомість, алгоритм пошуку 'розгортає' лише невелику його частину в процесі роботи, генеруючи наступні можливі стани (вершини) з поточного. Натомість, алгоритм пошуку генерує частини графа "на льоту", досліджуючи можливі дії з поточного стану.

Розв'язок задачі — це шлях (path) у цьому графі від початкової вершини (стану) до однієї з цільових вершин (станів). Якщо оператори мають вартість, то задача часто полягає у знаходженні не просто будь-якого шляху, а оптимального **шляху** (наприклад, найкоротшого або найдешевшого).

Мета алгоритму пошуку у просторі станів — систематично дослідити цей граф, щоб знайти шлях до мети. Різні алгоритми (як-от пошук у глибину, пошук у ширину, A*) є просто різними стратегіями дослідження цього графа.

Простір станів використовується у GPS-навігації та картографічних сервісах. Простір станів - це усі перехрестя та ключові точки на карті світу. Оператори - дороги, що

з'єднують ці точки. Ребра в цьому графі є зваженими (вага — це відстань або час у дорозі). основна задача - знайти найкоротший шлях між двома вершинами. Алгоритми, як-от A*, ефективно знаходять цей шлях. Вони роблять це, на кожному кроці обираючи для дослідження не просто найближчу точку, а ту, яка виглядає найбільш перспективною, враховуючи як вже пройдену відстань, так і приблизну відстань до мети. В ігровому ШІ (Game AI) простір станів - цсі можливі позиції на ігровій дошці. Оператори - всі легальні ходи для гравця. Необхідно знайти послідовність ходів, яка веде до виграшного стану. Алгоритм Minimax виконує рекурсивний пошук у глибину по дереву гри, щоб знайти оптимальний хід. На кожному рівні дерева він оцінює позиції, припускаючи, що ви (гравець 'Max') завжди будете обирати хід, що веде до максимальної оцінки, а ваш опонент ('Min') — до мінімальної. 'Піднімаючи' ці оцінки вгору по дереву, алгоритм знаходить найкращий хід для поточної позиції.
Задача пошуку у просторі станів використовується для планування дій для роботів, де простір станів - це усі можливі конфігурації робота та об'єктів у його оточенні (положення, орієнтація, стан маніпулятора). Оператори - це базові дії робота (рухатися_вперед, повернути_ліворуч, взяти_об'єкт). Головна мета - знайти послідовність дій (план), яка переведе робота з початкового стану (наприклад, "робот на базі") до цільового ("об'єкт доставлено у потрібне місце"). Наприклад, у класичній задачі "Світ блоків" (Blocks World), стан — це поточне розташування кубиків на столі, а оператори — це pick_up(A) та stack(A, B). Мета — знайти послідовність операторів, що приведе до заданої конфігурації кубиків.
Отже, пошук у просторі станів — це потужна концептуальна рамка, що дозволяє переформулювати різноманітні складні задачі в одну, добре вивчену проблему: пошук оптимального шляху в графі. Це фундаментальний підхід, який лежить в основі багатьох "інтелектуальних" можливостей: від здатності комп'ютера грати в шахи до вміння робота орієнтуватися в просторі. Розуміння цієї концепції є ключовим для розуміння того, як системи ШІ планують, приймають рішення та вирішують проблеми, особливо в умовах величезної кількості можливих варіантів.

6.3. Пошук в ширину і глибину як базові стратегії пошуку

Пошук шляху у лабіринті чи грі

Одним із найбільш наочних та класичних прикладів пошуку у просторі станів є задача знаходження шляху в лабіринті або на ігровій карті. Це завдання лежить в основі багатьох знайомих нам технологій: від інтелекту неігрових персонажів (NPC) у відеоіграх, що переслідують гравця, до роботів-пилососів, які планують оптимальний маршрут прибирання кімнати.
Суть задачі проста: знайти послідовність кроків від початкової точки до кінцевої, оминаючи перешкоди. Давайте розкладемо цю задачу на компоненти простору станів, щоб зрозуміти, як комп'ютер "бачить" цю проблему.
Лабіринт як простір станів

- Стан - це поточне місцезнаходження персонажа або агента. Зазвичай це координати (x, y) на двовимірній сітці.

- Початковий стан -це стартова клітинка лабіринту (наприклад, (0, 0)).
- Цільовий стан - клітинка з виходом (наприклад, (10, 10)).
- Оператори (дії) - можливі кроки з поточної клітинки. Зазвичай це рух вгору, вниз, вліво або вправо на сусідню клітинку, якщо вона не є стіною. Кожен такий крок має свою "вартість" (cost). У простому лабіринті вартість кожного кроку дорівнює 1. У складніших іграх вона може залежати від типу поверхні (наприклад, рух по піску "дорожчий", ніж по траві).
- Простір станів є уся сукупність клітинок лабіринту, по яких можна ходити.
- Розв'язок - це послідовність кроків (шлях) від початкового стану до цільового. Якщо дії мають вартість, то метою є знаходження оптимального шляху — шляху з найменшою сумарною вартістю.

Найприродніший спосіб представити лабіринт для комп'ютера — це граф, де кожна клітинка є вершиною, а можливість перейти до сусідньої клітинки — ребром.

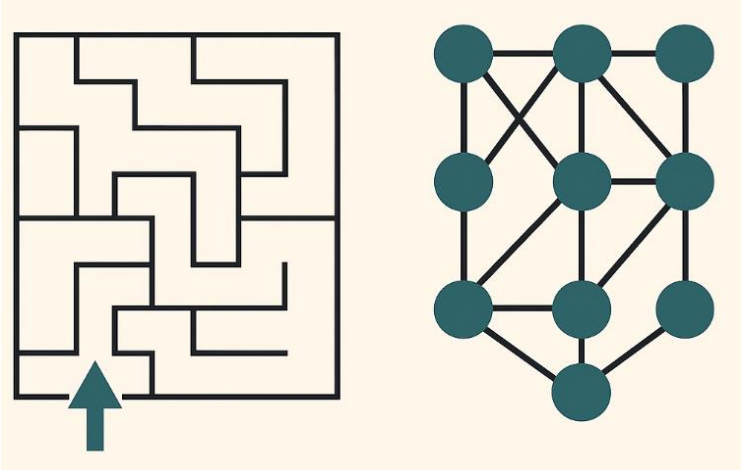


Рис.32. Лабіринт, представлений у вигляді сітки та відповідного графа

Для вирішення цієї задачі існують різні стратегії пошуку. Розглянемо дві найпопулярніші: пошук у ширину (BFS) та A* ("А-зірочка").

1. Пошук у ширину (Breadth-First Search, BFS)

BFS — це алгоритм, який систематично досліджує граф рівень за рівнем. Його можна назвати "обережним дослідником".

- Принцип роботи:
  1. Починає з початкового вузла.
  2. Відвідує всіх його безпосередніх сусідів (це "перший рівень").
  3. Потім відвідує всіх сусідів цих сусідів (це "другий рівень").
  4. І так далі, поки не знайде ціль.
- Аналогія: Уявіть, що ви кинули камінь у воду. Кола розходяться від центру рівномірно в усі боки. BFS працює так само, досліджуючи простір "колами", що розширюються. Він гарантовано спочатку знайде всі шляхи довжиною 1, потім всі шляхи довжиною 2, і так далі.
- Структура даних: Для відстеження вершин, які потрібно відвідати, BFS використовує чергу (Queue), що працює за принципом FIFO ("першим прийшов — першим пішов").

BFS завжди знаходить найкоротший шлях з точки зору кількості кроків (ребер), якщо вартість кожного кроку однакова, що є гарантією оптимальності. Він є "сліпим", що є

недоліком. Він досліджує всі можливі напрямки однаково, навіть ті, що очевидно ведуть у протилежний бік від мети. На великих відкритих картах це призводить до дослідження величезної кількості непотрібних вузлів і може бути дуже повільно.

Приклад реалізації на Python:

```
from collections import deque

def bfs(grid, start, goal):
    # У черзі зберігаємо повні шляхи
    queue = deque([[start]])
    # У множині visited зберігаємо вже відвідані клітинки, щоб не зациклитися
    visited = {start}

    while queue:
        path = queue.popleft()
        x, y = path[-1]

        if (x, y) == goal:
            return path # Шлях знайдено

        # Розглядаємо всіх сусідів: вгору, вниз, вліво, вправо
        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            next_x, next_y = x + dx, y + dy

            # Перевірка, чи не вийшли ми за межі поля, чи не є клітинка стіною (1)
            # і чи не відвідували ми її раніше
            if 0 <= next_x < len(grid) and 0 <= next_y < len(grid[0]) and \
               grid[next_x][next_y] != 1 and (next_x, next_y) not in visited:

                visited.add((next_x, next_y))
                new_path = list(path)
                new_path.append((next_x, next_y))
                queue.append(new_path)
    return None # Шлях не знайдено
```

2. **Пошук A* ("А-зірочка")** — це "розумний" алгоритм пошуку, який поєднує переваги BFS (знаходження найкоротшого шляху) та "жадібного" пошуку (рух у напрямку мети). Принцип роботи алгоритму A* наступний. А обирає, який вузол досліджувати наступним, на основі оціночної функції f(n), яка балансує між реальністю та припущенням:

f(n) = g(n) + h(n), де:

- g(n) — це реальна вартість шляху від початкового вузла до поточного n. Це точний, «песимістичний» компонент, що враховує вже зроблену роботу.
- h(n) — це евристична (приблизна) оцінка вартості шляху від n до цілі. Це «оптимістичне» припущення, яке спрямовує пошук. Для сітки часто використовується Манхеттенська відстань (|x1 - x2| + |y1 - y2|), що ігнорує перешкоди.

Прикладом може бути поїздка на машині з Києва до Львова. Ви не будете досліджувати всі дороги навколо Києва однаково. Ви оберете дорогу, яка веде на захід, тому що ви

знаєте, що Львів знаходиться саме там. h(n) — це ваше знання про напрямок, ваш "внутрішній компас".
Для відстеження вузлів, які потрібно відвідати, A* використовує структуру даних чергу з пріоритетом (Priority Queue), яка завжди повертає вузол з найменшим значенням f(n).
A* значно швидший за BFS на великих просторах, оскільки він цілеспрямовано рухається до мети, ігноруючи очевидно погані шляхи та гарантовано знаходить найкоротший шлях, якщо евристична функція h(n) є допустимою (admissible), тобто вона ніколи не переоцінює реальну вартість шляху до мети. Манхеттенська відстань є допустимою, оскільки вона представляє найкоротший можливий шлях на сітці без перешкод.

Приклад реалізації на Python (спрощений):

```
import heapq

def a_star(grid, start, goal):
    # У черзі з пріоритетом зберігаємо кортежі: (f_cost, g_cost, path)
    priority_queue = [(0, 0, [start])]
    visited = {start}

    while priority_queue:
        # Вилучаємо шлях з найменшим f_cost
        _, g_cost, path = heapq.heappop(priority_queue)
        x, y = path[-1]

        if (x, y) == goal:
            return path # Шлях знайдено

        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            next_x, next_y = x + dx, y + dy

            if 0 <= next_x < len(grid) and 0 <= next_y < len(grid[0]) and \
               grid[next_x][next_y] != 1 and (next_x, next_y) not in visited:

                visited.add((next_x, next_y))
                new_g_cost = g_cost + 1

                # Евристика (Манхеттенська відстань до мети)
                h_cost = abs(next_x - goal[0]) + abs(next_y - goal[1])
                f_cost = new_g_cost + h_cost

                new_path = list(path)
                new_path.append((next_x, next_y))

                # Додаємо новий шлях у чергу з пріоритетом
                heapq.heappush(priority_queue, (f_cost, new_g_cost, new_path))
    return None # Шлях не знайдено
```

Отже, алгоритми BFS та A* вважаються фундаментальними, оскільки вони представляють дві протилежні, але ключові філософії вирішення будь-якої задачі пошуку. BFS здійснює неінформований (сліпий), але систематичний пошук. Його можна

уявити як повінь, що рівномірно заповнює простір, поки не досягне мети. Це є базовий підхід, тому що він є еталоном вичерпного, але повного перебору. Якщо рішення існує, BFS його знайде, і це буде найкоротший шлях за кількістю кроків. Це найнадійніша „стратегія за замовчуванням“, коли про задачу нічого не відомо.
Алгоритм A здійснює інформований (евристичний) пошук.* На відміну від „повені“ BFS, пошук A* можна уявити як річку, що тече до моря. Вона завжди обирає шлях найменшого опору, керуючись гравітацією (евристикою), але при цьому може огинати перешкоди. Ц є базовий підхід, тому що він вводить ідею використання додаткових знань про проблему для значного прискорення пошуку. Це фундаментальний крок від „сліпого“ перебору до інтелектуального, цілеспрямованого вирішення.
Разом вони утворюють "два стовпи" пошукових алгоритмів: гарантована оптимальність, але повільне дослідження (BFS), проти швидкого, цілеспрямованого пошуку, який покладається на якість "підказок" (A*). Майже всі інші алгоритми пошуку шляху є або варіаціями цих двох, або їхньою комбінацією.

3. **Пошук в глибину (DFS)** — це базова стратегія обходу графа або дерева, яка досліджує структуру, занурюючись якомога глибше перед тим, як повернутися назад. Уяви, що ти досліджуєш лабіринт і завжди йдеш вперед, поки не увіпрешся в стіну — тоді повертаєшся назад і пробуєш інший шлях.

Алгоритм DFS наступний.

1. Починаємо з вибраної вершини (стартової).
2. Відвідуємо її та позначаємо як пройдену.
3. Рекурсивно переходимо до непройдених сусідів.
4. Якщо всі сусіди вже пройдені — повертаємося назад.
5. Повторюємо, поки не обійдемо всі досяжні вершини.

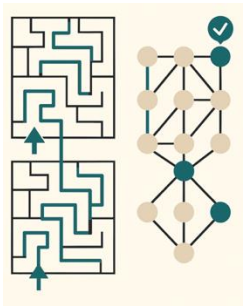


Рис.31. Пошук в глибину (DFS) на графі та лабіринті

В алгоритмі для збереження шляху використовується стек (stack) та множина відвіданих вершин — щоб уникнути повторного проходження.

Приклад на Python

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)  # або зберігаємо в список
```

```
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Приклад графа
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': ['F'],
    'F': []
}

dfs(graph, 'A')
```

Алгоритм пошуку в глибину (DFS) — це базовий метод обходу графів, застосовується для побудови дерев обходу, перевірки зв’язності, пошуку циклів, генерації та розв’язання лабіринтів. У сфері штучного інтелекту DFS використовується для планування, аналізу графів знань, перебору можливих станів та побудови дерев рішень. Його цінують за простоту, гнучкість і роль як складової частини складніших алгоритмів.

Контрольні запитання до розділу

1. Компроміс лінійного та бінарного пошуку. Бінарний пошук (O(log n)) є асимптотично набагато швидшим за лінійний (O(n)). Однак, у яких практичних ситуаціях використання "повільного" лінійного пошуку є більш доцільним та виправданим, ніж спроба застосувати бінарний?
2. "Крихкість" інтерполяційного пошуку. Інтерполяційний пошук в ідеальних умовах має вражаючу складність O(log log n). Чому, незважаючи на це, він не використовується як універсальний алгоритм пошуку в стандартних бібліотеках, на відміну від бінарного пошуку? Яка його головна слабкість?
3. Пошук у просторі станів як абстракція. Опишіть процес розв'язання головоломки (наприклад, кубика Рубіка або судоку) в термінах простору станів. Що в цьому випадку є. Станом? Початковим та цільовим станом? Операторами (діями)? Розміром простору станів (приблизно)?
4. Фундаментальна різниця між BFS та DFS. І Пошук у ширину (BFS), і Пошук у глибину (DFS) є алгоритмами повного перебору, які гарантовано знайдуть шлях, якщо він існує. Але вони знаходять принципово різні шляхи. Який з цих алгоритмів гарантує знаходження найкоротшого шляху (за кількістю кроків) і чому інший цієї гарантії не дає?
5. Пам'ять та стратегія. Як відрізняється використання пам'яті для BFS та DFS? Уявіть, що ви досліджуєте дуже "широкий", але "неглибокий" лабіринт. А потім — дуже

"глибокий", але "вузький" (з малою кількістю розгалужень). Який алгоритм (BFS чи DFS) ризикує вичерпати пам'ять у кожному з цих випадків і чому?

6. Роль структури даних. Робота BFS та DFS безпосередньо залежить від структури даних, яку вони використовують для зберігання вершин, що очікують на обробку (черга для BFS, стек для DFS). Що сталося б з порядком обходу, якби ви спробували реалізувати BFS, але замість черги використали б стек?
7. "Сліпота" пошуку. BFS та DFS називають "сліпими" або "неінформованими" алгоритмами пошуку. Що саме означає ця "сліпота"? Якої інформації їм не вистачає, яку, наприклад, використовує алгоритм A*?
8. Перехід від пошуку в масиві до пошуку шляху. Чим фундаментально відрізняється задача "знайти число 23 у відсортованому масиві" від задачі "знайти шлях до клітинки (2,3) у лабіринті"? Що є результатом пошуку в першому та другому випадку?
9. Вартість переходу. У класичному прикладі з лабіринтом вартість кожного кроку дорівнює 1. Як зміниться задача і чи буде BFS все ще знаходити оптимальний (найдешевший) шлях, якщо рух у певні клітинки (наприклад, "болото") буде "коштувати" 5 очок, а в звичайні — 1 очко?
10. Рефлексія про власні стратегії. Подумайте, як ви вирішуєте побутові задачі пошуку, наприклад, коли шукаєте загублені ключі в квартирі. Ваша стратегія більше схожа на BFS (систематично перевіряєте кімнату за кімнатою, не пропускаючи нічого) чи на DFS (глибоко перевіряєте одну зону, наприклад, кишені всього одягу, перш ніж перейти до іншої)?

Тести для самоконтролю

1. Яка з цих умов є обов'язковою для застосування бінарного пошуку?
    a) Масив повинен містити лише унікальні елементи.
    b) Кількість елементів у масиві має бути парною.
    c) Масив повинен бути відсортованим.
    d) Масив повинен містити лише цілі числа.
2. Яка структура даних використовується в основі алгоритму Пошуку в ширину (BFS) для відстеження вершин, що очікують на обробку?
    a) Черга (Queue)
    b) Стек (Stack)
    c) Хеш-таблиця
    d) Дерево
3. "Простір станів" у контексті ШІ — це:
    a) Конкретна конфігурація задачі в даний момент.
    b) Множина всіх можливих конфігурацій, у яких може перебувати задача.
    c) Послідовність дій для вирішення задачі.
    d) Найкоротший шлях від початку до мети.
4. Яка часова складність лінійного пошуку в найгіршому випадку?
    a) O(1)
    b) O(log n)
    c) O(n)
    d) $O(n^2)$

5. Який з цих алгоритмів гарантовано знайде найкоротший шлях (за кількістю кроків) у лабіринті, де кожен крок має однакову вартість?

a) Пошук у глибину (DFS)
b) Пошук у ширину (BFS)
c) Випадковий пошук
d) Лінійний пошук

6. У чому полягає головний недолік інтерполяційного пошуку, через який він не є універсальним?

a) Він вимагає занадто багато пам'яті.
b) Він працює повільніше за лінійний пошук.
c) Він не може працювати з відсортованими даними.
d) Його ефективність різко падає, якщо дані розподілені нерівномірно.

7. "Дослідити одну гілку графа на максимальну глибину, перш ніж повертатися назад" — це опис стратегії:

a) Пошуку A*
b) Пошуку в ширину (BFS)
c) Пошуку в глибину (DFS)
d) Бінарного пошуку

8. Задача знаходження оптимального маршруту в GPS-навігаторі — це класичний приклад:

a) Пошуку в масиві
b) Пошуку у просторі станів
c) Сортування даних
d) Кластеризації

9. В термінах простору станів, що таке "оператор" або "дія"?

a) Кінцева мета пошуку.
b) Унікальна конфігурація задачі.
c) Правило, що дозволяє перейти з одного стану в інший.
d) Вартість проходження шляху.

10. У вас є великий відсортований масив телефонних номерів (які є цілими числами, розподіленими досить рівномірно). Який алгоритм пошуку, швидше за все, буде найефективнішим?

a) Лінійний пошук
b) Бінарний пошук
c) Інтерполяційний пошук
d) Пошук у глибину (DFS)

Завдання для самостійної роботи

Це завдання допоможе вам глибше зрозуміти різницю між пошуком даних та пошуком рішень, а також на практиці реалізувати базові алгоритми обходу графа.

Частина 1: Аналіз та рефлекція (Письмово)

Дайте розгорнуті відповіді на наступні питання.

1. Ціна впорядкованості. Бінарний пошук є значно ефективнішим за лінійний, але вимагає попередньо відсортованих даних. Опишіть сценарій, у якому, незважаючи на великий розмір масиву, ви все одно обрали б лінійний пошук. Яке

співвідношення між частотою операцій пошуку та частотою додавання/зміни даних робить сортування невигідним?

2. Моделювання простору станів. Опишіть класичну задачу "Вовк, коза і капуста" (де фермеру потрібно перевезти всіх на інший берег річки, не залишаючи певні пари без нагляду) в термінах простору станів. Що в цій задачі є станом, початковим станом, цільовим станом та операторами (діями)?
3. Стратегії обходу графа. Поясніть ключову відмінність у стратегіях дослідження графа між Пошуком у ширину (BFS) та Пошуком у глибину (DFS). Чому BFS завжди знаходить найкоротший шлях за кількістю кроків, а DFS — ні? В якому випадку (наприклад, для якої структури лабіринту) DFS може знайти розв'язок значно швидше, ніж BFS?

Частина 2: Практичне завдання (Пошук шляху в лабіринті)

Напишіть скрипт на Python, який знаходить шлях у простому лабіринті.

1. Створіть лабіринт:
   Уявіть лабіринт у вигляді двовимірного масиву (списку списків), де 0 — це вільний шлях, а 1 — стіна. Наприклад:
   ```
   maze = [
      [0, 1, 0, 0, 0],
      [0, 1, 0, 1, 0],
      [0, 0, 0, 1, 0],
      [1, 1, 0, 1, 0],
      [0, 0, 0, 0, 0]
   ]
   ```
2. Визначте старт та фініш:
   Задайте початкову та кінцеву координати. Наприклад, start = (0, 0) та goal = (4, 4).
3. Реалізуйте алгоритм пошуку:
   - Напишіть функцію, яка реалізує Пошук у ширину (BFS) для знаходження шляху від старту до фінішу.
   - Функція повинна повертати список кортежів, що представляють координати шляху, або None, якщо шлях не існує.
4. Виведіть результат:
   - Якщо шлях знайдено, виведіть його на екран.
   - (За бажанням) Створіть функцію, яка візуалізує лабіринт, позначаючи знайдений шлях іншими символами (наприклад, *).

Частина 3: Дослідження та аналіз (Письмово)

У лекції BFS та DFS були названі "сліпими" або неінформованими алгоритмами пошуку. Їхньою протилежністю є інформовані алгоритми, найвідомішим з яких є A* ("А-зірочка").

1. Дослідіть та коротко опишіть своїми словами, у чому полягає основний принцип роботи алгоритму A*.
2. Що таке евристична функція h(n) в A*? Яку роль вона відіграє і чому робить пошук "розумнішим" порівняно з BFS?
3. Наведіть приклад допустимої евристичної функції, яку можна було б використати для задачі пошуку шляху в лабіринті на сітці. Поясніть, чому ця евристика є "допустимою" (admissible).

Розділ 7. Стратегії розроблення алгоритмів та погляд у майбутнє

Вступ

Досі ми розглядали окремі інструменти з арсеналу програміста: структури даних, які дозволяють ефективно організовувати інформацію, та базові алгоритми сортування і пошуку. Однак при зіткненні зі складною, нетривіальною проблемою, простого знання цих "цеглинок" недостатньо. Потрібен "архітектурний план" — загальна стратегія або підхід до вирішення задачі.

Саме тут на сцену виходять стратегії розробки алгоритмів. Це не конкретні алгоритми, а радше способи мислення та узагальнені методи, які можна застосувати до цілого класу проблем. Це набір потужних парадигм, що дозволяють розбити велику і незрозумілу задачу на менші, керовані частини і знайти елегантне та ефективне рішення там, де підхід "грубої сили" зазнав би поразки.

У цій темі ми перейдемо від вивчення інструментів до вивчення методів їх застосування. Ми дослідимо ключові стратегії, такі як "Розділяй та володарюй", "жадібні" алгоритми та динамічне програмування. Розуміння цих підходів перетворює розробку з хаотичного процесу на структуровану інженерну дисципліну і є тим, що відрізняє досвідченого програміста від початківця.

7.1. Стратегія "Розділяй і володарюй" на прикладі сортування злиттям

"Розділяй і володарюй" (Divide and Conquer) — це одна з найпотужніших та найфундаментальніших стратегій розробки алгоритмів. Замість того, щоб намагатися вирішити велику, складну проблему "в лоб", ця стратегія пропонує розбити її на менші, простіші підзадачі, вирішити їх окремо, а потім об'єднати їхні рішення, щоб отримати остаточну відповідь.

Уявіть, що вам потрібно порахувати кількість книг у величезній бібліотеці. Замість того, щоб ходити і рахувати кожну книгу самостійно, ви можете застосувати "Розділяй і володарюй":

1. Розділяй - ви просите двох помічників порахувати книги у лівій та правій половинах бібліотеки відповідно. Вони, у свою чергу, роблять те саме зі своїми помічниками, і так далі, поки кожен не отримає лише одну полицю для підрахунку.
2. Володарюй - порахувати книги на одній полиці — це дуже проста задача. Кожен виконує свою маленьку роботу.
3. Об'єднуй - кожен помічник повідомляє результат своєму керівнику. Керівники додають отримані числа і передають результат вище. Врешті-решт, ви отримаєте два числа від ваших головних помічників, додасте їх і отримаєте загальну кількість книг.

Цей процес складається з трьох канонічних етапів:

1. Розділення (Divide) - розбиття вихідної задачі на кілька менших підзадач того ж типу.
2. Володарювання (Conquer) - рекурсивне розв'язання цих підзадач. Якщо підзадачі достатньо малі, вони вирішуються напряму.
3. Об'єднання (Combine) - комбінування рішень підзадач для отримання рішення вихідної задачі.

Сортування злиттям (Merge Sort)— це класичний алгоритм сортування, який є втіленням стратегії "Розділяй і володарюй".

Задача: Відсортувати масив чисел [38, 27, 43, 3, 9, 82, 10].

1. Етап розділення (Divide)

Ми рекурсивно ділимо масив навпіл, поки не отримаємо підмасиви, що складаються з одного елемента. Масив з одного елемента за визначенням є відсортованим.

[38, 27, 43, 3, 9, 82, 10]
↓
[38, 27, 43, 3] та [9, 82, 10]
↓
[38, 27] та [43, 3] та [9, 82] та [10]
↓
[38] [27] та [43] [3] та [9] [82] та [10]

На цьому етапі розділення завершено. Ми маємо багато маленьких, тривіально вирішених підзадач (відсортованих масивів з одного елемента).

2. Етап володарювання та об'єднання (Conquer & Combine)

Тепер починається найважливіша частина — злиття (merge). Ми починаємо об'єднувати сусідні відсортовані підмасиви у більші відсортовані масиви, рухаючись знизу вгору.

1. Перший рівень злиття:
   - [38] і [27] -> [27, 38]
   - [43] і [3] -> [3, 43]
   - [9] і [82] -> [9, 82]
   - [10] залишається без змін.
2. Другий рівень злиття:
   - [27, 38] і [3, 43] -> [3, 27, 38, 43]
   - [9, 82] і [10] -> [9, 10, 82]
3. Фінальний рівень злиття:
   - [3, 27, 38, 43] і [9, 10, 82] -> [3, 9, 10, 27, 38, 43, 82]

Як працює сам процес злиття?

Уявіть, що у вас є два відсортовані масиви, наприклад, A = [27, 38] та B = [3, 43].

- Створюємо порожній результуючий масив.
- Порівнюємо перші елементи кожного масиву (27 та 3).
- 3 менше, тому беремо його і додаємо в результат. Тепер B виглядає як [43].
- Порівнюємо 27 та 43. 27 менше, беремо його. Тепер A виглядає як [38].
- Порівнюємо 38 та 43. 38 менше, беремо його. Масив A порожній.
- Додаємо в результат залишки з масиву B (тобто 43).
- Результат: [3, 27, 38, 43].

Реалізація на Python

```
def merge_sort(arr):
    # Базовий випадок рекурсії: масив з одного елемента вже відсортований
    if len(arr) <= 1:
        return arr
```

```
    # 1. Етап Розділення
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # 2. Етап Володарювання (рекурсивний виклик)
    sorted_left = merge_sort(left_half)
    sorted_right = merge_sort(right_half)

    # 3. Етап Об'єднання
    return merge(sorted_left, sorted_right)

def merge(left, right):
    result = []
    i = j = 0 # Індекси для лівого та правого підмасивів

    # Порівнюємо елементи і додаємо менший у результат
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Додаємо залишки з того масиву, який ще не закінчився
    result.extend(left[i:])
    result.extend(right[j:])

    return result

# Приклад використання
my_array = [38, 27, 43, 3, 9, 82, 10]
sorted_array = merge_sort(my_array)
print(f"Відсортований масив: {sorted_array}")
```

Отже, стратегія "Розділяй і володарюй" є надзвичайно потужною. Вона дозволяє звести складну задачу (сортування n елементів) до великої кількості простих операцій (злиття пар елементів), що в кінцевому підсумку призводить до дуже ефективного алгоритму з часовою складністю O(n log n**)**. Ця ж стратегія лежить в основі багатьох інших важливих алгоритмів, таких як швидке сортування (Quick Sort), бінарний пошук та алгоритми множення великих чисел.

7.2. Застосування стратегії "Розділяй і володарюй" для паралельних обчислень на GPU в системах ШІ

Сучасне глибоке навчання (Deep Learning) стикається з фундаментальною обчислювальною проблемою: як ефективно задіяти тисячі паралельних процесорних ядер для вирішення єдиної задачі. Рішення цієї проблеми лежить на перетині апаратних інновацій та застосування фундаментальної алгоритмічної стратегії — “Розділяй і володарюй”.

Саме ця концепція забезпечує необхідну методологічну основу для декомпозиції складних обчислювальних задач, уможливлюючи їх ефективне розпаралелювання та використання повної потужності графічних процесорів (GPU) для навчання великомасштабних моделей штучного інтелекту.

Стратегія "Розділяй і володарюй" передбачає декомпозицію вихідної задачі на множину менших, однотипних підпроблем. Ключовою вимогою є можливість незалежного розв'язання цих підпроблем, що є передумовою для їх паралельного виконання. Процес складається з трьох канонічних етапів:

1. Розділення (Divide): Початкова задача рекурсивно розбивається на підзадачі меншого розміру.
2. Володарювання (Conquer): Кожна підзадача розв'язується окремо. Якщо підзадача є достатньо простою, її рішення обчислюється напряму.
3. Об'єднання (Combine): Рішення, отримані на попередньому етапі, комбінуються для формування розв'язку вихідної задачі.

Незалежність підзадач на етапі "Володарювання" дозволяє уникнути послідовних залежностей, які могли б стати обмежуючим фактором для паралельної обробки.

Для розуміння важливості даної стратегії необхідно розглянути архітектурні відмінності між центральними (CPU) та графічними (GPU) процесорами. CPU (Центральний процесор) архітектурно оптимізований для мінімізації затримки (latency) при виконанні однієї складної, послідовної послідовності інструкцій. Він має невелику кількість потужних ядер, що ефективно справляються із задачами, які вимагають розгалуження логіки та прийняття рішень. GPU (Графічний процесор) архітектурно оптимізований для максимізації пропускної здатності (throughput). Він складається з тисяч значно простіших ядер, призначених для одночасного виконання однієї й тієї ж операції над великими масивами даних. Цей принцип функціонування відомий як SIMD (Single Instruction, Multiple Data).

Умовно, CPU можна порівняти з висококваліфікованим хірургом, що виконує одну складну операцію з мінімальною затримкою (low latency), тоді як GPU — це тисяча робітників на конвеєрі, які одночасно виконують тисячі простих, однотипних дій, забезпечуючи величезну пропускну здатність (high throughput).

Взаємодія між парадигмою "Розділяй і володарюй" та архітектурою GPU створює синергетичний ефект, що є основою продуктивності сучасних систем ШІ. Етап "Розділення" декомпозує великі обчислювальні задачі на мільйони однотипних незалежних операцій, що є ідеальним робочим навантаженням для GPU, який на етапі "Володарювання" виконує їх паралельно.

Розглянемо застосування цього принципу на прикладах ключових операцій.

1. Множення матриць

Множення матриць є центральною обчислювальною операцією в більшості архітектур нейронних мереж, оскільки саме воно реалізує процес поширення та трансформації сигналу (вектора ознак) від одного шару до іншого.

- Розділення - задача множення двох матриць A і B для отримання матриці C декомпозується на обчислення кожного окремого елемента C[i, j].
- Володарювання - обчислення кожного елемента C[i, j], що є скалярним добутком і-го рядка матриці A та j-го стовпця матриці B, є повністю незалежною підзадачею.
- Паралелізм на GPU - тисячі ядер GPU одночасно обчислюють тисячі різних елементів результуючої матриці C, реалізуючи принцип SIMD.
- Об'єднання - оскільки кожен елемент обчислюється і записується у відповідну позицію в пам'яті незалежно, етап об'єднання є тривіальним — результуюча матриця формується без додаткових операцій.

2. Згорткові шари (Convolutional Layers)

У згорткових нейронних мережах, що використовуються для аналізу зображень, застосовується фільтр (ядро) до різних ділянок вхідного зображення.

- Розділення - застосування фільтра до всього зображення розбивається на множину однакових операцій — застосування цього ж фільтра до кожної окремої ділянки зображення.
- Володарювання - кожна з цих операцій (поелементне множення та сумування) є незалежною від інших.
- Паралелізм на GPU - GPU одночасно застосовує той самий фільтр до сотень або тисяч різних ділянок вхідного зображення, що значно прискорює обчислення.

3. Обробка партій даних (Batch Processing)

Процес навчання моделей ШІ також підпорядковується цій стратегії. Моделі, як правило, навчаються не на одному прикладі за раз, а на "партіях" (batch) з десятків чи сотень зразків даних.

- Розділення - обробка всієї партії даних декомпозується на обробку кожного окремого зразка всередині цієї партії.
- Володарювання - пряме поширення (forward pass) для кожного зразка є незалежною обчислювальною задачею.
- Паралелізм на GPU - GPU може паралельно обробляти всі зразки з партії, виконуючи ті самі операції множення матриць для кожного з них. Такий підхід не тільки ефективно утилізує ресурси, але й стабілізує процес навчання, оскільки оновлення ваг моделі базується на усередненому градієнті по всій партії.

Отже, стратегія “Розділяй і володарюй” виступає не просто як один із багатьох алгоритмічних підходів, а як філософська основа, що уможливлює сучасні високопродуктивні обчислення. Вона слугує “мостом” між абстрактною математикою нейронних мереж та конкретною паралельною архітектурою графічних процесорів. Саме цей ідеальний симбіоз алгоритмічної декомпозиції та апаратної реалізації дозволив досягти експоненційного прогресу в галузі глибокого навчання, уможлививши створення та навчання моделей, складність яких ще десятиліття тому здавалася недосяжною.

7.3. Стратегія "Зменшуй і володарюй" на прикладі бінарного пошуку

Поряд із відомою стратегією "Розділяй і володарюй", існує інший, не менш потужний підхід до розробки алгоритмів — "Зменшуй і володарюй" (Decrease and Conquer). Хоча ці назви схожі, їхня філософія принципово різна. Якщо «Розділяй і володарюй» — це про делегування роботи кільком «помічникам» (обробка лівої та правої половин у сортуванні злиттям), то «Зменшуй і володарюй» — це про спрощення. На кожному кроці ми не створюємо нових завдань, а лише зменшуємо складність поточної, зводячи її до однієї, але меншої підзадачі. Уявіть, що ви чистите цибулину. Ви не розрізаєте її на частини. Ви знімаєте один шар, і перед вами залишається та сама цибулина, але трохи менша. Ви повторюєте цю дію, поки не дійдете до серцевини. Це і є суть стратегії "Зменшуй і володарюй".

Ця стратегія має три основні варіації:

1. Зменшення на константу. На кожному кроці розмір задачі зменшується на фіксоване число (зазвичай на 1). Класичний приклад — обчислення факторіала: n! = n * (n-1)!. Задача обчислення n! зводиться до меншої задачі обчислення (n-1)!.
2. Зменшення у певну кількість разів (на константний фактор). На кожному кроці розмір задачі зменшується у певну кількість разів (найчастіше — вдвічі). Це найпотужніша варіація, що призводить до логарифмічної складності.
3. Зменшення на змінну величину. Розмір зменшення залежить від властивостей задачі на поточному кроці. Приклад — алгоритм Евкліда для знаходження найбільшого спільного дільника, де на кожному кроці розмір чисел зменшується непередбачувано.

Бінарний пошук є канонічним прикладом другої варіації — зменшення у певну кількість разів.

Задача: Знайти індекс елемента у відсортованому масиві.

На відміну від сортування злиттям, нам не потрібно обробляти обидві половини масиву. Критична вимога до відсортованості даних дозволяє нам зробити однозначний висновок: якщо середній елемент не є тим, що ми шукаємо, то наш цільовий елемент може знаходитися тільки в одній з половин. Іншу половину можна повністю відкинути, не розглядаючи її взагалі.

Застосування стратегії:

1. Зменшуй - на кожному кроці ми порівнюємо шуканий елемент із середнім елементом масиву. На основі цього порівняння ми відкидаємо половину масиву. Вартість цього зменшення є мінімальною — лише одна операція порівняння. Саме ця комбінація — значне зменшення розміру задачі за дуже низьку ціну — робить алгоритм настільки ефективним. Таким чином, простір для пошуку (розмір задачі) зменшується вдвічі.
2. Володарюй - **п**ісля зменшення ми отримуємо нову, меншу підзадачу: "знайти елемент у новій, меншій половині масиву". Ми рекурсивно застосовуємо до неї ту саму логіку. "Володарювання" відбувається в той момент, коли підзадача стає тривіальною: або ми знайшли елемент, або простір для пошуку став порожнім, що означає відсутність елемента в масиві.

Крок за кроком на прикладі

Візьмемо відсортований масив: [2, 5, 8, 12, 16, 23, 38, 56, 72, 91] і шукаємо число 23.

- Крок 1: Початкова задача
  - Простір для пошуку: весь масив (індекси 0-9), розмір 10.

- Середній індекс: (0 + 9) // 2 = 4.
- Порівнюємо 23 із середнім елементом arr[4] = 16.
- Оскільки 23 > 16, ми робимо висновок, що шуканий елемент, якщо він існує, може знаходитися лише праворуч від 16. Ліва половина відкидається.

- Крок 2: Зменшена підзадача
  - Зменшуємо простір для пошуку до правої половини (індекси 5-9), розмір 5.
  - Новий середній індекс: (5 + 9) // 2 = 7.
  - Порівнюємо 23 із середнім елементом нового простору arr[7] = 56.
  - Оскільки 23 < 56, ми відкидаємо праву частину цього підмасиву.
- Крок 3: Ще менша підзадача
  - Зменшуємо простір для пошуку до [23, 38] (індекси 5-6), розмір 2.
  - Новий середній індекс: (5 + 6) // 2 = 5.
  - Порівнюємо 23 із середнім елементом arr[5] = 23.
  - Елементи рівні. Задача вирішена.

Ми не розбивали задачу на кілька частин для окремої обробки. Ми послідовно зменшували одну й ту саму задачу, поки не знайшли рішення.

Реалізація на Python (ітеративна)

```
def binary_search(sorted_arr, target):
    low = 0
    high = len(sorted_arr) - 1

    while low <= high:
        # Поточна "задача" - це діапазон від low до high
        mid = low + (high - low) // 2 # Цей варіант обчислення є більш безпечним у мовах, де може статися переповнення цілого числа (integer overflow), якщо `low` та `high` дуже великі. В Python це не є проблемою, але це гарна практика, про яку варто знати.
        guess = sorted_arr[mid]

        if guess == target:
            return mid # "Володарювання": задача вирішена, підпроблема стала тривіальною

        # "Зменшення": обираємо, яку половину відкинути, і звужуємо діапазон
        elif guess > target:
            high = mid - 1 # Нова, менша задача - це ліва половина
        else:
            low = mid + 1  # Нова, менша задача - це права половина

    return -1 # Задача не має рішення, оскільки простір пошуку порожній

# Приклад використання
my_array = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
target = 23
result = binary_search(my_array, target)
print(f"Індекс елемента {target}: {result}")
```

Отже, стратегія «Зменшуй і володарюй» є потужною парадигмою для розв'язання задач, де можна знайти відношення між рішенням вихідної задачі та рішенням її меншої версії.

Бінарний пошук є яскравим прикладом, де зменшення простору вдвічі на кожному кроці призводить до «логарифмічного стрибка» в ефективності. Однак варто пам'ятати про її головний компроміс: для роботи вона вимагає певної структури даних (у цьому випадку — відсортованого масиву), підготовка якої сама по собі є обчислювально затратною. Розуміння цієї стратегії та її компромісів є ключовим для вибору правильного алгоритму.

7.4. Вступ до ітеративних стратегій. Градієнтний спуск

Досі ми розглядали алгоритмічні стратегії, які гарантовано знаходять точне, оптимальне рішення (якщо воно існує). Сортування злиттям завжди ідеально відсортує масив, а пошук A* знайде найкоротший шлях. Це аналітичні або прямі методи. Однак існує величезний клас задач, особливо у сфері штучного інтелекту, де знайти точне аналітичне рішення є неможливим або обчислювально надто складним.
Як навчити нейронну мережу з мільярдами параметрів розпізнавати зображення? Як знайти найкращу лінію регресії для мільйона точок даних? Не існує простої формули, яка б видала нам правильні відповіді за один крок.
Для вирішення таких проблем використовуються ітеративні стратегії. Замість того, щоб намагатися знайти відповідь за один раз, ці стратегії починають з певного початкового припущення і крок за кроком, ітерація за ітерацією, покращують його, наближаючись до оптимального рішення. Вони не обіцяють знайти ідеал, але гарантують поступове вдосконалення.
Найважливішим та найпоширенішим ітеративним алгоритмом в сучасному ШІ є градієнтний спуск (Gradient Descent). Це робочий кінь, що лежить в основі навчання майже всіх моделей глибокого навчання.

Концептуальне пояснення градієнтного спуску

Уявіть, що ви стоїте на вершині гори в густому тумані і хочете якнайшвидше спуститися вниз, у найнижчу точку долини. Ви не бачите всієї картини, але можете відчути нахил поверхні прямо під ногами.
Якою буде ваша стратегія?
Ви подивитеся навколо себе і визначите напрямок, у якому схил є найкрутішим вниз. Потім ви зробите невеликий крок у цьому напрямку. Опинившись у новій точці, ви знову повторите процес: знайдете найкрутіший спуск і зробите ще один крок. Повторюючи цей процес, крок за кроком, ви врешті-решт дійдете до найнижчої точки.
Це і є інтуїтивне пояснення градієнтного спуску.
Переклад на мову математики та ШІ:

- Гора (або "ландшафт помилок") - це функція втрат (loss function). Вона показує, наскільки сильно помиляється наша модель при поточних налаштуваннях. Уявіть, що цей ландшафт існує не в 3D, а в просторі з мільярдами вимірів, де кожна вісь — це один параметр (вага) нейронної мережі. Чим вище ми на "горі", тим більша помилка. Цей багатовимірний ландшафт є надзвичайно складним, з безліччю локальних западин, плато та ущелин. Саме тому знайти аналітичний розв'язок (тобто 'побачити' всю карту одразу) неможливо, і потрібен ітеративний підхід.
- Найнижча точка долини - це мінімум функції втрат, точка, де помилка моделі є найменшою. Це і є наша мета.

- Поточне місцезнаходження - це поточні значення параметрів (ваг) моделі. Це точка з мільярдами координат у просторі помилок.
- Напрямок найкрутішого спуску - це антиградієнт. Градієнт — це вектор, що складається з часткових похідних функції втрат по кожному параметру. Кожен компонент цього вектора показує, як сильно зміниться помилка, якщо трохи змінити відповідну вагу. Вектор градієнта завжди вказує у напрямку найшвидшого зростання функції. Відповідно, вектор, що вказує у протилежному напрямку (мінус градієнт), вказує на напрямок найшвидшого спадання.
- Розмір кроку - це швидкість навчання (learning rate), гіперпараметр, що контролює, наскільки сильно ми змінюємо ваги моделі на кожному кроці.

Ітеративна стратегія "Зроби маленький крок у бік правильної відповіді"

Градієнтний спуск є втіленням цієї простої, але потужної ітеративної ідеї. Процес навчання нейронної мережі виглядає так:

1. Ініціалізація. Ми починаємо з випадкових значень ваг моделі (опиняємося у випадковій точці на "горі помилок").
2. Ітеративний цикл (епоха за епохою):
   a. Розрахунок прогнозу. Подаємо на вхід моделі партію даних і отримуємо її прогнози.
   b. Обчислення помилки. Порівнюємо прогнози з реальними значеннями і обчислюємо значення функції втрат (наскільки високо ми зараз на "горі").
   c. Обчислення градієнта: За допомогою алгоритму зворотного поширення помилки (backpropagation), який є ефективним способом застосування ланцюгового правила диференціювання для всіх шарів мережі, обчислюємо градієнт — вектор, що показує, як зміна кожної ваги вплине на помилку.
   d. Крок у правильному напрямку. Коригуємо кожну вагу моделі, роблячи невеликий крок у напрямку, протилежному до градієнта. Розмір цього кроку регулюється швидкістю навчання.
   нова_вага = стара_вага - швидкість_навчання * градієнт
3. Повторення. Повертаємося до кроку (a) і повторюємо цей процес тисячі або мільйони разів, поки помилка не перестане суттєво зменшуватися.

З кожною ітерацією модель робить маленький крок "вниз по схилу", поступово зменшуючи свою помилку і наближаючись до оптимального набору ваг.

Ідеї ітеративного покращення застосовуються не лише в градієнтному спуску. Ось кілька прикладів з різних галузей ШІ.

1. Лінійна регресія за допомогою градієнтного спуску

Хоча для простої лінійної регресії існує аналітичне рішення, градієнтний спуск є чудовим прикладом ітеративного підходу.

- Задача. Знайти найкращу пряму ($y = mx + c$), що проходить через набір точок.
- Параметри: Нахил m та зсув c.
- Ітеративний процес:
  1. Ініціалізація. Починаємо з випадкових m та c (малюємо випадкову пряму).
  2. Обчислення помилки. Рахуємо, наскільки далеко наші точки знаходяться від цієї прямої (наприклад, за допомогою середньоквадратичної помилки).
  3. Крок. Обчислюємо градієнт помилки по відношенню до m та c і трохи змінюємо їх, щоб "повернути" пряму ближче до точок.
  4. Повторення. Повторюємо крок 3, і з кожною ітерацією пряма буде все краще і краще "лягати" на дані.

2. Кластеризація. Алгоритм K-середніх (K-Means)

Це популярний алгоритм для групування даних без попередньої розмітки.

- Задача. Розділити N точок даних на K груп (кластерів).

- Ітеративний процес:
    1. Ініціалізація. Випадковим чином обираємо K точок, які стають початковими "центрами" кластерів (центроїдами).
    2. Крок присвоєння. Кожну точку даних відносимо до того кластера, центр якого до неї найближчий.
    3. Крок оновлення. Перераховуємо позицію кожного центроїда як середнє арифметичне всіх точок, що тепер належать до його кластера.
    4. Повторення. Повторюємо кроки 2 і 3. З кожною ітерацією центроїди "мігрують" до справжніх центрів скупчень даних, а межі кластерів стають все більш стабільними. Процес зупиняється, коли позиції центроїдів перестають змінюватися.

3. Рекомендаційні системи: Матрична факторизація

Як Netflix рекомендує вам фільми? Один з підходів — матрична факторизація, яка також використовує ітеративну оптимізацію.

- Задача. Заповнити пропуски у величезній матриці "користувач-фільм", де відомі лише деякі оцінки.
- Ітеративний процес:
    1. Ініціалізація. Припускаємо, що смаки кожного користувача та характеристики кожного фільму можна описати невеликим набором прихованих факторів (наприклад, "комедійність", "драматичність"). Ми починаємо з випадкових значень для цих факторів.
    2. Обчислення прогнозу. На основі цих випадкових факторів прогнозуємо оцінки для тих фільмів, які користувачі вже оцінили.
    3. Обчислення помилки. Порівнюємо наші прогнози з реальними оцінками.
    4. Крок: Використовуючи варіант градієнтного спуску (наприклад, SGD), ми трохи змінюємо приховані фактори для користувачів та фільмів, щоб зменшити помилку.
    5. Повторення. Повторюючи цей процес, модель ітеративно "вивчає" справжні вподобання користувачів та характеристики фільмів, що дозволяє їй робити все точніші прогнози для ще не переглянутих фільмів.

Вибір правильного розміру кроку (learning rate) є критично важливим:

- якщо занадто великий крок, ми можемо "перестрибнути" мінімум і почати "метатися" по схилах гори, так і не досягнувши дна. У такому випадку помилка може коливатися або навіть зростати. Процес навчання буде нестабільним.
- Якщо занадто маленький крок, спуск буде дуже повільним і може зайняти надто багато часу, або ми можемо "застрягти" у невеликій локальній западині (локальному мінімумі), так і не діставшись до глобального мінімуму.

Сучасні алгоритми оптимізації (як-от Adam, RMSprop) вирішують цю проблему, використовуючи адаптивну швидкість навчання, яка автоматично коригує розмір кроку для кожного параметра окремо протягом усього процесу навчання.

Отже, ітеративні стратегії, і зокрема градієнтний спуск, є фундаментальним механізмом оптимізації, що лежить в основі практично всього сучасного глибокого навчання. Вони перетворюють нерозв'язну задачу пошуку аналітичного мінімуму в складній функції на послідовність керованих кроків. Замість того, щоб шукати ідеальну відповідь за один раз, ми починаємо з грубого припущення і поступово, крок за кроком, покращуємо його. Цей підхід, хоч і не завжди гарантує знаходження абсолютно найкращого рішення (можна потрапити в локальний мінімум), на практиці виявився надзвичайно ефективним для навчання найскладніших нейронних мереж, що лежать в основі сучасних технологій, від розпізнавання облич до великих мовних моделей.

7.5. Жадібні алгоритми та динамічне програмування

Жадібні алгоритми та динамічне програмування є фундаментальними стратегіями розроблення алгоритмів. Вони стоять в одному ряду з такими підходами, як "розділяй та володарюй", алгоритми з поверненням та метод гілок та меж. Обидві стратегії використовуються для розв'язання оптимізаційних задач, однак роблять це принципово різними шляхами.

1. Жадібні алгоритми (Greedy Algorithms)

Це алгоритмічна стратегія, яка на кожному кроці робить вибір, що видається найкращим у даний момент (локально оптимальним), сподіваючись, що послідовність таких виборів призведе до глобально оптимального рішення.

Ключові характеристики:

- **Принцип:** "Бери найкраще, що можеш отримати зараз".
- **Простота:** Зазвичай простіші в реалізації та швидші за інші методи.
- **Недалекоглядність:** Алгоритм не переглядає зроблені раніше вибори і не оцінює їхні довгострокові наслідки.
- **Обмежена оптимальність:** Жадібний підхід гарантує знаходження найкращого рішення лише для певного класу задач (що мають "властивість жадібного вибору"). Для багатьох інших проблем він може давати неоптимальні результати.

Аналогія: Уявіть, що ви спускаєтеся з вершини гори і на кожному кроці обираєте найкрутіший схил вниз. Ця стратегія швидко приведе вас до якоїсь долини, але не обов'язково до найнижчої точки в усьому гірському масиві.

Класичні приклади жадібних алгоритмів:

Алгоритм Дейкстри: Пошук найкоротшого шляху у графі. Жадібний вибір: обрати найближчу ще не відвідану вершину.

Задача: Знайти найкоротший шлях від однієї (стартової) вершини до всіх інших вершин у зваженому графі, де ваги ребер не є від'ємними.

Жадібний підхід: На кожному кроці алгоритм обирає ще не відвідану вершину, до якої веде найкоротший шлях від стартової. Ця вершина додається до множини відвіданих, і відстані до її сусідів оновлюються.

- Локально оптимальний вибір. Вибрати вершину з найменшою поточною відстанню від початкової.
- Чому це працює? Оскільки ваги ребер невід'ємні, якщо ми знайшли найкоротший шлях до вершини V, то будь-який інший шлях до неї (через інші, ще не відвідані вершини) буде гарантовано довшим.

Застосування: GPS-навігація, мережева маршрутизація, пошук найшвидшого маршруту в комп'ютерних іграх.

Алгоритми Крускала та Прима: Побудова мінімального кістякового дерева. Жадібний вибір: додати до дерева найлегше ребро, що не утворює циклу.

Задача. Знайти мінімальне кістякове (остовне) дерево у зв'язному, неорієнтованому, зваженому графі. Мінімальне кістякове дерево — це підграф, який з'єднує всі вершини разом з мінімально можливою сумарною вагою ребер, не утворюючи циклів.

Жадібний підхід: Алгоритм розглядає всі ребра графа, відсортовані за зростанням їхньої ваги. На кожному кроці він бере найлегше ребро, яке ще не розглядалося, і додає його до дерева, якщо воно не утворює циклу з уже доданими ребрами.

- Локально оптимальний вибір. Вибрати найлегше ребро, яке не створює циклу.
- Чому це працює? Додаючи найдешевші ребра, ми гарантуємо, що загальна вага буде мінімальною. Правило уникнення циклів забезпечує те, що ми будуємо саме дерево.

Застосування. Проектування мереж (телекомунікаційних, комп'ютерних, транспортних), кластеризація даних.

Алгоритм Хаффмана: Стиснення даних. Жадібний вибір: об'єднати два символи з найменшими частотами.
Задача. Стиснення даних. Побудувати префіксний код для набору символів, що має мінімальну очікувану довжину кодового слова. Символам, які зустрічаються частіше, надаються коротші коди.
Жадібний підхід. Алгоритм починає з набору вузлів, кожен з яких представляє символ та його частоту. На кожному кроці він знаходить два вузли з найменшими частотами, об'єднує їх у новий батьківський вузол (чия частота дорівнює сумі частот нащадків) і замінює ці два вузли на новий. Процес повторюється, доки не залишиться один кореневий вузол.

- Локально оптимальний вибір. Об'єднати два символи (або піддерева) з найнижчими частотами.
- Чому це працює? Розміщуючи найрідкісніші символи глибше в дереві, алгоритм гарантує, що найчастіші символи матимуть найкоротші кодові шляхи, що мінімізує загальну довжину закодованого повідомлення.

Застосування. Архіватори (наприклад, GZIP), формати файлів (JPEG, PNG, MP3), протоколи стиснення даних в інтернеті.

Задача про вибір заявок (Activity Selection Problem)
Задача. Є набір заявок (активностей), кожна з яких має час початку та час закінчення. Потрібно вибрати максимальну кількість заявок, які не перетинаються в часі, щоб їх можна було виконати на одному ресурсі (наприклад, в одній аудиторії).
Жадібний підхід. Відсортувати всі заявки за часом їх закінчення. Потім вибрати першу заявку в цьому списку. Далі послідовно переглядати решту заявок і обирати першу, яка починається не раніше, ніж закінчилася попередня обрана.

- Локально оптимальний вибір. Вибрати заявку, яка закінчується найраніше.
- Чому це працює? Обираючи заявку, що закінчується якомога раніше, ми звільняємо ресурс для наступних можливих заявок раніше, тим самим максимізуючи шанси виконати більше заявок.

2. Динамічне програмування (Dynamic Programming) — це методологія розроблення алгоритмів для розв'язання складних оптимізаційних задач шляхом їх розбиття на простіші, менші підзадачі, що перетинаються.
Ключові характеристики методології наступні. Принцип дії "Розділяй, розв'язуй, запам'ятовуй і комбінуй". Задача має властивість оптимальної підструктури, якщо її глобально оптимальний розв'язок може бути побудований з оптимальних розв'язків її підзадач. Під час розв'язання виникає потреба багаторазово обчислювати розв'язки одних і тих самих підзадач. Щоб уникнути повторних обчислень, результати розв'язання підзадач зберігаються в пам'яті (наприклад, у масиві чи хеш-таблиці). Динамічне програмування розв'язує кожну підзадачу лише один раз. Для задач, що задовольняють його вимогам, динамічне програмування завжди гарантує знаходження глобально оптимального рішення.

Як приклад: щоб знайти найкоротший шлях до 10-ї сходинки, ви спершу знаходите і запам'ятовуєте найкоротші шляхи до 1-ї, 2-ї, 3-ї і так далі. Щоб дістатися до 10-ї сходинки, ви просто робите один крок з тієї попередньої сходинки (8-ї чи 9-ї), з якої шлях був найкоротшим.

3. Порівняння та ключові відмінності

Хоча обидва підходи застосовуються до задач, що мають властивість оптимальної підструктури, ключова відмінність лежить у способі прийняття рішень. Жадібний підхід робить один локально оптимальний вибір і рухається далі. Динамічне програмування розглядає всі можливі варіанти на кожному кроці, щоб гарантувати глобально оптимальний вибір.

***Таблиця** 19. Порівняльна таблиця основних характеристик жадібних алгоритмів та динамічного програмування*

Характеристика	**Жадібні алгоритми**	**Динамічне програмування**
Основний принцип	Робить локально оптимальний вибір на кожному кроці.	Знаходить оптимальні розв'язки для підзадач і використовує їх для побудови загального розв'язку.
Гарантія оптимальності	Не завжди гарантує знаходження глобального оптимуму.	Завжди гарантує знаходження глобального оптимуму, якщо задача має оптимальну підструктуру.
Прийняття рішень	Робить один вибір і рухається далі. Рішення є остаточним.	Розглядає декілька варіантів вибору, обираючи найкращий на основі розв'язків підзадач.
Швидкодія	Зазвичай швидші, оскільки не потребують розв'язання всіх підзадач.	Можуть бути повільнішими через необхідність обчислення та збереження результатів для багатьох підзадач.
Складність реалізації	Як правило, простіші в реалізації.	Часто складніші для розуміння та імплементації.
Підхід до розв'язання	Зазвичай "згори-вниз" (top-down).	Класично "знизу-вгору" (bottom-up), хоча можлива і рекурсивна реалізація з меоізацією.

Наочний приклад. Задача про здачу (Coin Change Problem)

Розглянемо задачу видачі здачі мінімальною кількістю монет. Припустимо, у нас є монети номіналами {1, 5, 10, 25} і нам потрібно видати 30 копійок.

- Жадібний алгоритм діятиме так:
    - Беремо найбільшу можливу монету: 25. Залишається 5.
    - Беремо найбільшу можливу монету для залишку: 5. Залишається 0.
    - Результат: 2 монети (25 + 5). Це оптимальний розв'язок.

А тепер змінимо набір номіналів на {1, 8, 10} і нам потрібно видати 16 копійок.

- Жадібний алгоритм:
    - Беремо найбільшу можливу монету: 10. Залишається 6.
    - Для 6 найбільша монета — 1. Беремо 6 монет по 1.
    - Результат: 7 монет (10 + 1 + 1 + 1 + 1 + 1 + 1). Це не оптимально!
- Динамічне програмування знайде всі проміжні оптимальні розв'язки і визначить, що 16 = 8 + 8.
    - Результат: 2 монети. Це оптимальний розв'язок.

Цей приклад наочно демонструє, чому жадібний підхід не завжди працює і чому динамічне програмування є більш надійним, хоч і обчислювально складнішим методом. Спільне вивчення дозволяє чітко окреслити межі застосовності кожного підходу.

4. Застосування в штучному інтелекті

Обидві стратегії є будівельними блоками для багатьох технологій в ШІ.

Жадібні алгоритми в ШІ цінуються за швидкість та ефективні евристики:

- Побудова дерев рішень (CART): На кожному вузлі приймається жадібне рішення про найкраще розділення даних.
- Апроксимація для NP-складних задач: Використовуються для швидкого пошуку наближених рішень (напр., вибір ознак, задача комівояжера).
- Генерація тексту в NLP (Greedy Search): На кожному кроці обирається наступне слово з найвищою ймовірністю.

Динамічне програмування в ШІ є основою для гарантовано оптимальних рішень:

- Навчання з підкріпленням (Reinforcement Learning): Є теоретичною основою для розв'язання Марковських процесів прийняття рішень (алгоритми ітерації за політиками та цінностями).
- Обробка природної мови (NLP): Алгоритм Вітербі використовує ДП для знаходження найімовірнішої послідовності прихованих станів (напр., розмітка частин мови).
- Біоінформатика та комп'ютерний зір: Вирівнювання послідовностей ДНК (алгоритм Нідлмана-Вунша), побудова карт глибини.

Отже, у світі ШІ жадібні алгоритми та динамічне програмування не конкурують, а доповнюють один одного. Жадібні алгоритми обирають, коли потрібна швидкість і "достатньо добре" рішення є прийнятним. Динамічне програмування використовується, коли необхідний гарантовано оптимальний розв'язок для складної задачі, що розбивається на підзадачі.

Контрольні запитання до розділу

1. "Розділяй" проти "Зменшуй". У чому полягає фундаментальна концептуальна різниця між стратегіями "Розділяй і володарюй" (на прикладі сортування злиттям) та "Зменшуй і володарюй" (на прикладі бінарного пошуку)? Як ця різниця впливає на кількість підзадач, які потрібно вирішити на кожному кроці?
2. Зв'язок з апаратним забезпеченням. Як саме стратегія "Розділяй і володарюй" уможливлює ефективні паралельні обчислення на сучасних графічних процесорах (GPU)? Чому ця стратегія є настільки важливою для тренування великих моделей штучного інтелекту?

3. Ітеративні стратегії та оптимальність. Прямі алгоритми, як-от сортування, знаходять точне, гарантовано правильне рішення. Ітеративні алгоритми, як-от градієнтний спуск, лише наближаються до рішення. В яких випадках використання ітеративних стратегій є не просто виправданим, а єдиним можливим підходом?
4. Аналогія з градієнтним спуском. Повертаючись до аналогії зі спуском з гори в тумані, що в цій аналогії представляли б такі поняття. Занадто велика швидкість навчання (learning rate)? Локальний мінімум? Стохастичний градієнтний спуск (SGD)?
5. Жадібна стратегія (Greedy Approach). Жадібна стратегія полягає в тому, щоб на кожному кроці робити локально оптимальний вибір, сподіваючись, що це приведе до глобально оптимального рішення. Наведіть приклад задачі, де такий підхід спрацює, і приклад, де він гарантовано призведе до неправильного результату.
6. Вибір правильної стратегії. Уявіть, що вам потрібно розробити алгоритм для системи GPS-навігації, щоб знайти найшвидший маршрут між двома містами. Яка з вивчених стратегій (Розділяй і володарюй, Зменшуй і володарюй, Ітеративна/Пошук у просторі станів) є найбільш відповідною для цієї задачі і чому?
7. Динамічне програмування. Стратегія динамічного програмування вирішує задачу, розбиваючи її на підзадачі, що перекриваються, і зберігаючи результати цих підзадач, щоб не обчислювати їх повторно (мемоізація). Чим це принципово відрізняється від "Розділяй і володарюй", де підзадачі зазвичай є незалежними?
8. Зв'язок стратегій та структур даних. Як вибір структури даних впливає на можливість застосування певної стратегії? Наприклад, чому бінарний пошук (стратегія "Зменшуй і володарюй") вимагає відсортованого масиву? Чи можна було б застосувати його до зв'язного списку з тією ж ефективністю?
9. Практичний наслідок. Уявіть, що ви працюєте над моделлю ШІ, і процес її навчання зупинився на певному рівні помилки і більше не покращується. Використовуючи знання про градієнтний спуск, які можуть бути три найімовірніші причини такої поведінки і які ваші перші кроки для вирішення проблеми?
10. Рефлексія та застосування в житті. Подумайте про складний проєкт або задачу з вашого особистого життя (наприклад, вивчення нової навички, планування великого заходу). Спробуйте описати ваш підхід до її вирішення в термінах вивчених стратегій. Чи використовуєте ви "Розділяй і володарюй", розбиваючи проєкт на частини? Чи дієте ітеративно, роблячи маленькі кроки і коригуючи курс?

Тести для самоконтролю

1. Яка стратегія розробки алгоритмів найкраще описує сортування злиттям (Merge Sort)?
   a) Жадібна стратегія
   b) Ітеративна стратегія

c) Розділяй і володарюй
d) Зменшуй і володарюй

2. Бінарний пошук є класичним прикладом якої стратегії?
   a) Розділяй і володарюй
   b) Динамічне програмування
   c) Зменшуй і володарюй
   d) Метод повного перебору
3. Що є "напрямком найкрутішого спуску" в аналогії з градієнтним спуском?
   a) Градієнт
   b) Антиградієнт (мінус градієнт)
   c) Швидкість навчання
   d) Функція втрат
4. Яка ключова відмінність стратегії "Зменшуй і володарюй" від "Розділяй і володарюй"?
   a) "Зменшуй і володарюй" завжди працює швидше.
   b) "Зменшуй і володарюй" зводить задачу до однієї меншої підзадачі, а "Розділяй і володарюй" — до кількох.
   c) "Розділяй і володарюй" використовує рекурсію, а "Зменшуй і володарюй" — ні.
   d) "Зменшуй і володарюй" працює тільки на відсортованих даних.
5. Чому стратегія "Розділяй і володарюй" ідеально підходить для паралельних обчислень на GPU?
   a) Тому що GPU має дуже швидкі ядра.
   b) Тому що вона розбиває задачу на безліч незалежних підзадач, які можна виконувати одночасно.
   c) Тому що вона вимагає багато пам'яті, яка є у GPU.
   d) Тому що вона завжди знаходить аналітичне рішення.
6. Що станеться, якщо обрати занадто велику швидкість навчання (learning rate) при градієнтному спуску?
   a) Навчання буде дуже повільним.
   b) Модель гарантовано знайде глобальний мінімум.
   c) Алгоритм завершиться за одну ітерацію.
   d) Процес навчання може стати нестабільним, а помилка — коливатися або навіть зростати.
7. Яка операція в сортуванні злиттям відповідає етапу "Об'єднання" (Combine)?
   a) Розбиття масиву навпіл.
   b) Злиття двох відсортованих підмасивів в один.
   c) Вибір опорного елемента.
   d) Рекурсивний виклик для однієї з половин.
8. Яка варіація градієнтного спуску обчислює градієнт по одному випадковому прикладу за раз?
   a) Міні-партіонний градієнтний спуск
   b) Повний градієнтний спуск
   c) Стохастичний градієнтний спуск (SGD)
   d) Адаптивний градієнтний спуск
9. Який з перелічених алгоритмів не є прикладом стратегії "Розділяй і володарюй"?
   a) Швидке сортування (Quick Sort)
   b) Сортування злиттям (Merge Sort)
   c) Сортування включенням (Insertion Sort)
   d) Алгоритм множення Карацуби

10. У чому полягає основна мета ітеративних стратегій?
    a) Знайти точне аналітичне рішення за один крок.
    b) Завжди знаходити глобально оптимальне рішення.
    c) Послідовно покращувати початкове припущення, наближаючись до оптимального рішення.
    d) Розбити задачу на незалежні підзадачі.

Завдання для самостійної роботи

Це завдання допоможе вам глибше зрозуміти фундаментальні відмінності між основними алгоритмічними стратегіями та їхнє практичне застосування у вирішенні складних задач, зокрема в ШІ.

Частина 1: Аналіз та рефлексія (Письмово)

Дайте розгорнуті відповіді на наступні питання.

1. "Розділяй" проти "Зменшуй". Сортування злиттям (Merge Sort) є прикладом стратегії "Розділяй і володарюй", а бінарний пошук — "Зменшуй і володарюй". Поясніть, у чому полягає ключова різниця в тому, як ці дві стратегії працюють з підзадачами. Чому одна призводить до рекурсивних викликів для кількох частин, а інша — лише для однієї?
2. Ітеративні стратегії та оптимальність. Алгоритми, засновані на "Розділяй і володарюй" (як Merge Sort), зазвичай знаходять точне, оптимальне рішення. Ітеративні алгоритми (як градієнтний спуск) лише наближаються до оптимального рішення. Поясніть, чому для такої задачі, як навчання нейронної мережі, ми змушені використовувати ітеративний підхід, а не можемо застосувати прямий, аналітичний метод?
3. Градієнтний спуск та його виклики. Уявіть "ландшафт помилок" для нейронної мережі. Опишіть своїми словами, що таке локальний мінімум і чому "застрягання" в ньому є однією з головних проблем градієнтного спуску. Як, на вашу думку, варіації алгоритму (наприклад, стохастичний градієнтний спуск) можуть допомогти частково вирішити цю проблему?

Частина 2: Практичне завдання (Реалізація градієнтного спуску)

Напишіть скрипт на Python, який реалізує найпростіший градієнтний спуск для знаходження мінімуму параболи $f(x) = x^2$.

1. Напишіть функцію. Створіть функцію f(x), яка повертає x**2, та функцію df(x), яка повертає її похідну (градієнт), тобто 2*x.
2. **Реалізуйте алгоритм:**
    - Ініціалізуйте початкову точку (наприклад, x = 10).
    - Задайте швидкість навчання (learning rate, наприклад, 0.1) та кількість ітерацій (наприклад, 20).
    - Напишіть цикл, який на кожній ітерації:
        - Обчислює градієнт у поточній точці x.
        - Оновлює x за формулою градієнтного спуску: x = x - learning_rate * gradient.
        - Виводить на екран номер ітерації, поточне значення x та значення функції f(x).

3. **Проведіть експеримент:**
   - Запустіть ваш скрипт. Простежте, як значення x поступово наближається до 0 (де знаходиться мінімум функції x^2).
   - Спробуйте змінити швидкість навчання (наприклад, на 0.9, а потім на 0.001) і подивіться, як це вплине на процес збіжності. Опишіть свої спостереження у коментарях.

Частина 3: Дослідження та аналіз (Письмово)

Стратегія "Розділяй і володарюй" лежить в основі не лише сортування злиттям, а й **Швидкого сортування (Quick Sort)**.

1. Дослідіть та коротко опишіть, як саме стратегія "Розділяй і володарюй" реалізована в алгоритмі Quick Sort.
2. Що в цьому алгоритмі відповідає етапам "Розділення", "Володарювання" та "Об'єднання"? (Підказка: етап об'єднання може бути тривіальним).
3. Порівняйте підхід до "розділення" у Merge Sort та Quick Sort. Чим вони принципово відрізняються і як це впливає на вимоги до пам'яті та продуктивність у найгіршому випадку?

Лабораторні роботи та проекти

Лабораторна робота №1

Тема: "Підготовка даних: базовий аналіз та очищення"

Мета: Отримати базові навички роботи з файлами, рядками та простими структурами даних (списками), що є основою для будь-якої задачі з аналізу даних.

AI-контекст: Data Scientists стверджують, що до 80% їхнього часу йде на збір та очищення даних (Data Cleaning & Preparation). Ця робота імітує цей найважливіший перший крок.

Завдання:

1. Створити текстовий файл з даними (напр., Прізвище, Рік_народження, Оцінка). Деякі рядки мають бути навмисно пошкоджені (неправильний формат, текстові значення замість чисел).
2. Написати програму, яка читає цей файл.
3. Для кожного рядка: перевірити коректність даних. Якщо дані коректні, прочитати їх в одному форматі (зазвичай текстовому) і перетворити у структурований вигляд, який програма може зрозуміти та використати їх та додати до списку зі словників.
4. Обчислити та вивести просту статистику: середній вік та середню оцінку для валідних записів.

Лабораторна робота №2

Тема: "Операції над матрицями: аналіз ігрової карти"

Мета: Відпрацювати навички створення та обробки двовимірних масивів (матриць) за допомогою вкладених циклів.

AI-контекст: Робота з матрицями — це основа комп'ютерного зору та лінійної алгебри, на якій побудовані нейронні мережі. Також це базова логіка для будь-якої стратегічної гри або симуляції.

Завдання:

1. Представити ігрову карту або "теплову карту" у вигляді двовимірного масиву 10x10, заповненого випадковими числами.
2. Реалізувати функції для:
    - Знаходження координат та значення максимального елемента ("найгарячіша точка").
    - Обчислення суми значень у заданому квадраті (напр., з [2,2] по [5,5]).
    - "Нормалізації" матриці: приведення всіх значень до діапазону від 0 до 1 шляхом ділення на максимальний елемент.

Проект 1: "Консольний 'Photoshop': основи обробки зображень"

AI-контекст: Будь-яка модель комп'ютерного зору (від розпізнавання облич у телефоні до автопілота) працює із зображеннями як з матрицями пікселів. Цей проект дає студентам відчути дані "на дотик" на найнижчому рівні.

Основні структури та алгоритми:

- Двовимірні масиви (матриці).
- Цикли та умовні оператори.
- Робота з файлами (для читання та запису спрощеного формату зображень).

Опис завдання:

- Написати програму, яка читає "зображення" з простого текстового файлу, де кожен рядок — це пікселі, розділені пробілами (наприклад, у градаціях сірого від 0 до 255).
- Зберегти це зображення у двовимірному масиві.
- Реалізувати декілька функцій-фільтрів:
    - Збільшення/зменшення яскравості (додати/відняти константу до кожного елемента).
    - Конвертація в чорно-біле зображення (значення > 128 стають 255, решта — 0).
    - Горизонтальне/вертикальне віддзеркалення.
    - Зберегти результат в новий файл.
- "Крок уперед" (необов'язково). Реалізувати простий фільтр розмиття (blur), де значення кожного пікселя усереднюється зі значеннями його сусідів.

Лабораторна робота №3

Тема: "Аналіз ефективності: порівняння алгоритмів сортування"

Мета: На практиці побачити різницю в часовій складності алгоритмів та зрозуміти важливість вибору ефективного алгоритму.

AI-контекст: При роботі з великими даними (Big Data) в машинному навчанні різниця між алгоритмами складності O(n2) та O(nlogn) може означати різницю між годинами та секундами обчислень.

Завдання:

1. Реалізувати два алгоритми сортування: один простий (напр., сортування обміном або "бульбашкою") та один більш ефективний (напр., сортування злиттям або швидке сортування).
2. Згенерувати масиви різного розміру (напр., 100, 1000, 10000 елементів).
3. За допомогою модуля time виміряти час виконання кожного алгоритму на кожному з масивів.
4. Представити результати у вигляді таблиці та написати висновок, що пояснює отримані результати.

Лабораторна робота №4

Тема: "Аналізатор тексту: підрахунок частоти слів"

Мета: Опанувати роботу зі словниками (хеш-таблицями) для швидкого пошуку, підрахунку та агрегації даних.

AI-контекст: Частотний аналіз тексту — це базовий крок у переважній більшості задач обробки природної мови (Natural Language Processing, NLP), від аналізу тональності до машинного перекладу.

Завдання:

1. Взяти невеликий текстовий файл (напр., статтю з Вікіпедії).
2. Написати програму, що читає текст, приводить його до нижнього регістру та видаляє основні знаки пунктуації.
3. Використовуючи словник, підрахувати, скільки разів у тексті зустрічається кожне слово.
4. Вивести на екран 10 найпопулярніших слів та їхню частоту.

Проект 2: "Мій перший класифікатор: Дерево рішень"

AI-контекст: Дерева рішень — один з найпопулярніших та найзрозуміліших алгоритмів класичного машинного навчання. Студенти створюють спрощену модель, яка може робити передбачення.

Основні структури та алгоритми:

1. Дерева (представлені через вкладені словники або об'єкти).
2. Словники (Dictionaries) та Записи (Records) для зберігання даних.
3. Рекурсія для обходу дерева.

Опис завдання:

1. Взяти простий набір даних (наприклад, 10-15 записів у форматі CSV або просто у коді). Приклад: погода, температура, вологість -> чи йти гуляти?.
2. Створити функцію, яка на основі цих даних будує просте дерево рішень (можна навіть "захардкодити" логіку на основі аналізу даних).
3. Написати функцію predict, яка приймає нові дані (наприклад, сонячно, тепло, сухо) і, проходячи по вузлах дерева, повертає фінальне рішення ("так, іти гуляти").

"Крок уперед" (необов'язково). Спробувати реалізувати простий алгоритм автоматичної побудови дерева, наприклад, на основі критерію "найбільшого приросту інформації" для одного рівня глибини.

Лабораторна робота №5

Тема: "Соціальна мережа: пошук найкоротшого шляху"

Мета: Навчитись представляти графи у вигляді списків суміжності та реалізовувати алгоритм пошуку в ширину (BFS).

AI-контекст: Аналіз соціальних графів, моделювання мереж та пошук зв'язків — ключові задачі, що вирішуються за допомогою теорії графів.

Завдання:

1. Представити невелику соціальну мережу (5-7 людей) у вигляді словника (списку суміжності).
2. Реалізувати алгоритм пошуку в ширину (BFS), використовуючи чергу (можна імітувати за допомогою списку в Python).
3. Програма має приймати на вхід двох людей (імена) і знаходити найкоротший ланцюжок знайомств між ними (напр., "Анна -> Богдан -> Віктор").

Проект 3: "Розумний кур'єр: пошук найкоротшого шляху"

AI-контекст: Пошук оптимального шляху — це класична задача штучного інтелекту, що лежить в основі GPS-навігаторів, логістики та розробки відеоігор.

Основні структури та алгоритми:

1. Графи (представлені через матрицю суміжності або списки суміжності).
2. Черги (Queues) для реалізації алгоритму пошуку в ширину (BFS).
3. Алгоритми пошуку (BFS або Дейкстри).

Опис завдання:

1. Створити представлення карти міста у вигляді графу, де вузли — це перехрестя, а ребра — дороги між ними. Ребрам можна надати "вагу" (довжину вулиці).
2. Користувач вводить початкову та кінцеву точку.
3. Програма повинна реалізувати алгоритм пошуку (для початку BFS, якщо всі дороги рівні, або Дейкстри для доріг різної довжини) для знаходження найкоротшого шляху.
4. Вивести знайдений шлях як послідовність перехресть.

"Крок уперед" (необов'язково). Реалізувати більш "розумний" алгоритм A* (A-star), додавши евристичну функцію (наприклад, пряму відстань до фінішу).

Алфавітний покажчик

А

Абстракція, рівні (в контексті Python) Це концепція, яка описує, наскільки сильно мова програмування приховує від розробника низькорівневі деталі роботи комп'ютера. Python має високий рівень абстракції над пам'яттю. Замість вказівників (змінних, що зберігають адресу в пам'яті, як у C++), він використовує посилання. У Python змінна — це не "коробка" для даних, а "ярлик", що посилається на об'єкт у пам'яті.

АВЛ-дерево (AVL Tree) Це один з перших і найсуворіших видів збалансованого бінарного дерева пошуку. Його ключова особливість полягає в тому, що для кожного вузла висота його лівого та правого піддерев може відрізнятися не більше ніж на одиницю. Якщо ця умова порушується, дерево автоматично "балансує" себе за допомогою операцій повороту, гарантуючи, що часова складність пошуку, вставки та видалення завжди залишається O(log n).

Алгоритм Це скінченна послідовність чітко визначених, виконуваних кроків, розроблена для вирішення певної задачі або виконання обчислення.

Алгоритми пошуку Це загальна категорія алгоритмів, призначених для знаходження певного елемента (або елементів) у структурі даних. Прикладами є лінійний, бінарний пошук (для масивів) та пошук у ширину (BFS), пошук у глибину (DFS) (для графів).

Алгоритми сортування Це загальна категорія алгоритмів, що використовуються для впорядкування елементів у колекції за певним критерієм. Прикладами є бульбашкове, злиттям, швидке сортування.

Антиградієнт Це вектор, протилежний до градієнта. У контексті оптимізації, якщо функція втрат представляє собою "ландшафт помилок", антиградієнт вказує на напрямок найшвидшого спадання помилки. Саме в цьому напрямку алгоритм градієнтного спуску робить крок на кожній ітерації.

Б

Балансування дерев Це процес підтримки бінарного дерева пошуку у стані, де висота лівого та правого піддерев для кожного вузла є приблизно однаковою, щоб уникнути виродження дерева у зв'язний список і зберегти логарифмічну складність операцій.

Бінарний пошук (Binary Search) Це високоефективний алгоритм (O(log n)) для знаходження елемента у відсортованому масиві, який на кожному кроці відкидає половину простору для пошуку. Є прикладом стратегії "Зменшуй і володарюй".

Бінарне дерево пошуку (BST) Це вузлова деревна структура, де для кожного вузла всі значення в його лівому піддереві є меншими, а в правому — більшими.

Бульбашкове сортування (Bubble Sort) Це класичний, але неефективний ($O(n^2)$) алгоритм сортування, що багаторазово проходить по масиву, порівнюючи та міняючи місцями сусідні елементи.

В

Вектор ознак (Feature Vector) Це одновимірний масив (1D тензор), який представляє один об'єкт у числовому вигляді для моделі машинного навчання.

Видалення (з масивів) Одна з базових операцій опрацювання масивів, що полягає у вилученні елементів (рядків або стовпців) для очищення даних або зменшення розмірності.

Вказівники (Pointers) Концепція з мов типу C++, де змінна зберігає адресу в пам'яті. У Python абстрагована за допомогою посилань та автоматичного керування пам'яттю.

Вокабуляр (словник унікальних слів) У NLP — це набір усіх унікальних слів (токенів), що зустрічаються в тексті. Для його створення використовуються множини або хеш-таблиці.

Г

Граф (Graph) Це нелінійна структура даних, що складається з вершин (вузлів) та ребер (зв'язків), яка моделює складні мережеві зв'язки.

Градієнтний спуск (Gradient Descent) Це основний ітеративний алгоритм оптимізації в ШІ, що мінімізує функцію втрат шляхом кроків у напрямку антиградієнта.

Д

"Розділяй і володарюй" (Divide and Conquer) Це алгоритмічна стратегія, яка розбиває задачу на кілька менших незалежних підзадач, вирішує їх рекурсивно, а потім об'єднує результати.

Двовимірний масив (2D Array) Структура даних, що організовує елементи у вигляді сітки (рядки та стовпці); також відома як матриця.

Двозв'язний список (Doubly Linked List) Тип зв'язного списку, де кожен вузол має посилання як на наступний, так і на попередній елемент.

Дек (Deque) Двостороння черга (Double-Ended Queue), що дозволяє ефективно додавати та видаляти елементи з обох кінців.

Дерево (Tree) Ієрархічна структура даних, що складається з вузлів та ребер без циклів.

Дерево рішень (Decision Tree) Простий та інтерпретований алгоритм класифікації та регресії у вигляді деревної структури правил.

Е

Евристика (Heuristic) "Освічене припущення" або правило, що використовується в інформованих алгоритмах пошуку (як-от A*) для оцінки відстані до мети та спрямування пошуку.

Ефективність алгоритмів Оцінка ресурсів (часу, пам'яті), які потрібні алгоритму, зі збільшенням розміру вхідних даних. Зазвичай описується за допомогою O-нотації.

Ж

"Жадібна" стратегія (Greedy Strategy) Алгоритмічна парадигма, що на кожному кроці робить локально оптимальний вибір, сподіваючись досягти глобального оптимуму.

З

"Зменшуй і володарюй" (Decrease and Conquer) Алгоритмічна стратегія, яка зводить задачу до однієї меншої підзадачі того ж типу (наприклад, бінарний пошук).

Зв'язний список (Linked List) Лінійна структура даних, що складається з вузлів, які посилаються один на одного; є альтернативою масивам.

І

Ітеративні стратегії (Iterative Strategies) Підхід до вирішення задач шляхом послідовного наближення до розв'язку, починаючи з початкового припущення (наприклад, градієнтний спуск).

Інтерполяційний пошук (Interpolation Search) Алгоритм пошуку для відсортованих та рівномірно розподілених даних, що робить "освічене припущення" про позицію елемента.

К

k-NN (k-найближчих сусідів) Алгоритм машинного навчання, який класифікує об'єкт на основі класів його k найближчих сусідів, що вимагає сортування відстаней.
Кешування (Caching) Техніка зберігання результатів "дорогих" операцій (зазвичай у хеш-таблиці) для уникнення повторних обчислень.
Класифікація (Classification) Задача машинного навчання, метою якої є присвоєння об'єкту однієї з попередньо визначених категорій (класів).
Колізія (в хеш-таблицях) Ситуація, коли хеш-функція генерує однаковий індекс для двох або більше різних ключів.
Кортеж (Tuple) Незмінний (immutable) тип даних у Python для зберігання впорядкованих послідовностей.
Керування пам'яттю Процес виділення та звільнення ресурсів пам'яті. У Python він автоматизований, на відміну від мов типу C++.

Л
Лінійний пошук (Linear Search) Базовий алгоритм пошуку в масивах, що полягає у послідовному переборі елементів.

М
Масив (Array) Структура даних, що зберігає набір елементів у неперервному блоці пам'яті. Основа для векторів та матриць.
Матриця (Matrix) Двовимірний масив (сітка з рядками та стовпцями).
Мемоізація (Memoization) Техніка кешування результатів функцій для уникнення повторних обчислень.
Множина (Set) Невпорядкована колекція унікальних елементів, реалізована на хеш-таблиці.
Модулі (Modules) Файли з кодом Python (.py), що дозволяють повторно використовувати функції, класи та змінні.

Н
Нейронна мережа (Neural Network) Обчислювальна модель, структуру якої можна представити у вигляді зваженого спрямованого графа.
NumPy Ключова бібліотека Python для наукових обчислень, що надає об'єкт ndarray.

О
О-нотація (Big O notation) Математична нотація для оцінки асимптотичної ефективності алгоритмів.
Обробка даних (попередня) (Data Preprocessing) Етап підготовки "сирих" даних для навчання моделі ШІ.
Одновимірний масив (1D Array) Масив з одним виміром (вектор), що використовується для представлення векторів ознак.
Однозв'язний список (Singly Linked List) Базова форма зв'язного списку з рухом лише в одному напрямку.

П
Pandas Бібліотека Python для аналізу табличних даних, що надає структуру DataFrame.
Паралельні обчислення (Parallel Computing) Одночасне виконання багатьох обчислень, що є ключовим для роботи GPU та навчання моделей ШІ.
Перестановка (в масивах) Операція зміни порядку елементів, що використовується, наприклад, для перемішування (shuffling) датасету.

Піраміда (Heap) Деревоподібна структура даних, що є основою для пірамідального сортування та черги з пріоритетом.
Пірамідальне сортування (Heapsort) Ефективний алгоритм сортування (O(n log n)), що працює "на місці".
Порозрядне сортування (Radix Sort) Лінійний не порівняльний алгоритм сортування для цілих чисел.
Посилання (в Python) Модель роботи зі змінними в Python, де змінна є "ярликом", що вказує на об'єкт.
Пошук (в масиві) Загальна назва для базових алгоритмів знаходження елемента у масиві.
Пошук в ширину (BFS) Стратегія обходу графа, що досліджує його рівень за рівнем.
Пошук в глибину (DFS) Стратегія обходу графа, що досліджує одну гілку на максимальну глибину.
Пошук у просторі станів (State Space Search) Концепція пошуку рішень в ШІ шляхом представлення задачі у вигляді графа станів.
Програма Конкретна реалізація алгоритму певною мовою програмування.
PyTorch Фреймворк для глибокого навчання, де тензор є ключовим об'єктом.

Р
Ранжування (Ranking) Застосування сортування в ШІ для впорядкування елементів за релевантністю.
Рекурсія (Recursion) Техніка, за якої функція викликає саму себе; основа для обходу дерев та графів.
Рекомендаційні системи Системи ШІ, що моделюють інтереси за допомогою графів для надання рекомендацій.

С
Словник (Dictionary) Вбудований тип даних Python (реалізація хеш-таблиці) для зберігання пар "ключ-значення".
Сортування (Sorting) Загальна тема, що охоплює алгоритми для впорядкування елементів.
Сортування вибором (Selection Sort) Простий квадратичний ($O(n^2)$) алгоритм сортування.
Соціальні мережі Онлайн-платформи, що моделюються за допомогою графів.
Список (List) Базовий змінний (mutable) впорядкований тип даних у Python.
Стабільність сортування (Sorting Stability) Властивість алгоритму сортування не змінювати відносний порядок однакових елементів.
Стек (Stack) Структура даних, що працює за принципом LIFO (Last-In, First-Out).

Т
Тензор (Tensor) Ключовий об'єкт у фреймворках глибокого навчання, що є узагальненням масивів та підтримує GPU/Autograd.
TensorFlow Фреймворк для глибокого навчання від Google.

Ф
Файли, робота з Набір базових команд для взаємодії з даними у файловій системі.
Функціональне програмування Парадигма програмування, що розглядає обчислення як оцінку математичних функцій.

Х

Хеш-таблиця (Hash Table) Ключовий інструмент для швидкого доступу до даних (O(1)), основа для словників та множин.
Хеш-функція (Hash Function) Функція, що перетворює ключ на індекс у масиві; основа роботи хеш-таблиць.

Ч
Червоно-чорне дерево (Red-Black Tree) Приклад збалансованого бінарного дерева пошуку.
Черга (Queue) Структура даних, що працює за принципом FIFO (First-In, First-Out).
"Чорна скринька" (Black Box) Поняття, що описує складні, неінтерпретовані моделі ШІ.

Ш
ШІ (Штучний інтелект) Скрізна тема посібника; галузь, що створює системи, здатні до інтелектуальних завдань.

Список літератури

1. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. Cambridge, MA: MIT Press, 2022.
2. Goodrich M. T., Tamassia R., Goldwasser M. H. Data Structures and Algorithms in Python. Hoboken, NJ: Wiley, 2013.
3. Skiena S. S. The Algorithm Design Manual. 3rd ed. New York: Springer, 2020.
4. McDowell G. L. Cracking the Coding Interview: 189 Programming Questions and Solutions. 6th ed. Palo Alto, CA: CareerCup, 2015.
5. Burkov A. The Hundred-Page Machine Learning Book. [S.l.], 2019.
6. Géron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. 3rd ed. Sebastopol, CA: O'Reilly Media, 2022.
7. Chollet F. Deep Learning with Python. 2nd ed. Shelter Island, NY: Manning Publications, 2021.
8. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge, MA: MIT Press, 2016.
9. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 4th ed. Boston: Pearson, 2020.
10. Barabási A.-L. Network Science. Cambridge: Cambridge University Press, 2016.
11. Aggarwal C. C. Recommender Systems: The Textbook. New York: Springer, 2016.
12. Deisenroth M. P., Faisal A. A., Ong C. S. Mathematics for Machine Learning. Cambridge: Cambridge University Press, 2020.
13. Kochenderfer M. J., Wheeler T. A. Algorithms for Optimization. Cambridge, MA: MIT Press, 2019.
14. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol, CA: O'Reilly Media, 2017.
15. Slatkin B. Effective Python: 90 Specific Ways to Write Better Python. 2nd ed. Boston: Addison-Wesley Professional, 2019.
16. Ammar W. AI Engineers Handbook [Electronic resource]. Conceptual publication.
17. Vaswani A., Shazeer N., Parmar N., et al. Attention Is All You Need // Advances in Neural Information Processing Systems. 2017.
18. T. Nosenko. AI-First: Algorithms and Data Structures as the Foundation of AI [Електронний ресурс]. Amazon KDP, 2025. URL: https://bit.ly/4omC346
19. Бхаргава А. Грокаємо алгоритми. Ілюстрований посібник для програмістів і не тільки / пер. з англ. — Харків: Фабула, 2019. — (Серія «#PROCreators»).
20. Трофименко О. Г., Прокоп Ю. В., Швайко І. Г. та ін. Алгоритми та структури даних: навч. посіб. — Одеса: Фенікс, 2019.
21. Креневич А. П. Алгоритми та структури даних: підручник. — Київ: ВПЦ «Київський університет», 2020.
22. Ярмокіна Т. В. Аналіз даних та машинне навчання мовою Python: практичний посібник. — Київ: КПІ ім. Ігоря Сікорського, 2021. — [Електронний ресурс]. Доступно в електронному архіві бібліотеки КПІ.